LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no.559-564
cop. 2

Digitized by the Internet Archive
in 2013

http://archive.org/details/modularmicroprog564kete

510.84
I26r
no.564
cop.2

Math

UIUCDCS-R-73-564

A MODULAR MICROPROGRAMMED COMPUTER WITH CONCURRENT DECENTRALIZED CONTROL

by

Mark Loren Ketelsen

January, 1973

THE LIBRARY OF THE
MAR 7 1973
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-73-564

A MODULAR MICROPROGRAMMED COMPUTER WITH CONCURRENT DECENTRALIZED CONTROL

by

Mark Loren Ketelsen

January, 1973

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

This work was supported in part by the National Science Foundation under Grant No. US NSF GJ-36265 and was submitted in partial fulfillment for the Master of Science degree in Electrical Engineering, 1973.

ACKNOWLEDGMENT

The author wishes to express gratitude to his thesis advisor Professor James E. Robertson. Assistant Professor James E. Vander Mey deserves a special thanks for his direct supervision and many helpful discussions throughout the research reported in this thesis. Professor Don Gillies has also been very helpful.

The author is deeply indebted to the Electrical Engineering Department and University of Illinois fellowship committees, and to the Department of Computer Science for financial support.

The dedicated work of Mrs. June Wingler in preparing this manuscript is gratefully acknowledged.

TABLE OF CONTENTS

Page

1. INTRODUCTION..... 1

2. SYSTEM PLANNING AND ORGANIZATION..... 2

2.1 Design Objectives, and Design Overview..... 2

2.2 Multibus Organization and Bus Operations..... 5

2.3 Instruction Stream Generation and Distribution..... 7

2.4 Bus Devices and Instruction Types..... 19

2.4.1 Arithmetic and Logical Unit..... 22

2.4.2 Shift Unit..... 24

2.4.3 Branch Control Unit..... 27

2.4.4 Inter-Bus Communication..... 34

2.4.5 Scratch-Pad Memory..... 37

2.4.6 Main Memory..... 40

2.4.7 External Interface..... 42

2.5 Multiple Bus Controllers..... 43

2.6 Implementation Considerations..... 44

3. SYSTEM SIMULATION..... 50

3.1 Purpose of Simulation..... 50

3.2 Simulator Implementation..... 50

3.3 Theoretical Performance Simulation Results..... 50

3.4 VK-1 Logical Simulator..... 52

Page

4. MICROPROGRAMMING VK-1 56

4.1 Firmware Development 56

4.2 Symbolic Microprogramming Language 59

4.3 Emulation of the IBM System/360 on VK-1 60

4.3.1 VK-1 Hardware used in 360 Emulation 61

4.3.2 Emulator Organization 61

4.3.3 Bus Assignment for Emulator Microprogram 68

4.3.4 Emulator Performance Analysis 71

5. CONCLUSIONS 78

LIST OF REFERENCES 79

APPENDIX

A. THEORETICAL PERFORMANCE SIMULATION PROGRAM 80

B. LOGICAL SYSTEM SIMULATION PROGRAM 83

C. IBM SYSTEM/360 EMULATOR MICROPROGRAM 97

LIST OF FIGURES

Figure		Page
2.1	Microinstruction Format	4
2.2	Simplified Block Diagram of VK-1	4
2.3	Bus Cycle Timing Diagram	8
2.4	Bus Controller Flow Diagram	9
2.5	Cycle-Initiate Controller for Control Store Module i	11
2.6	Instruction Stream Generator	13
2.7	Instruction Strobe Controller	14
2.8	A Typical Sequence of Activity in the Sequence Controller	16
2.9	Immediate Data Logic	18
2.10	Source Bus Controller	20
2.11	Destination Bus Controller	21
2.12	Arithmetic and Logic Unit	23
2.13a	Shifter Unit Block Diagram	28
2.13b	Shifter Unit Control Format	29
2.14a	Example Code for Branch	32
2.14b	Instruction Queue of Sequence Controller During Execution of Branch with Post-instructions	33
2.15	Branch Control Unit	35
2.16	BCU Command Format	36
2.17	Auto-increment Scratch Memory Unit	39
2.18	Scratch Memory Controller Algorithms	41

Figure Page

2.19 Multiple Bus Control Unit Configuration..... 45

2.20a Bus Arbitration Controller..... 46

2.20b Bus Device Interface Controller..... 46

2.20c Instruction Strobe Controller..... 47

2.20d Bus Master Flag Controller..... 47

3.1 Model for Theoretical Performance Simulation..... 51

3.2 Execution Time vs. Number of Busses..... 53

3.3 Execution Time vs. Taccess..... 54

4.1 Bus Assignment for Example..... 58

4.2 Overlap of Bus Cycle for Example..... 58

4.3 Scratch Memory Assignment..... 62

4.4 System/360 Emulator Flow Chart..... 65

4.5 Flow of Control Through Emulator..... 70

LIST OF TABLES

Table		Page
2.1	Bus Signatures	6
2.2	Control Store Module Control Signals	10
2.3	ALU Operation Set	25
2.4	ALU Condition Flags	26
4.1	Bus Register Utilization	63
4.2	Device Mnemonics used in Emulator	69
4.3	360 Emulator Bus Assignment	72

1. INTRODUCTION

This thesis presents and evaluates the structure of a microprogram-controlled digital computer called VK-1. In this processor, highly encoded microinstructions are executed by a set of functional modules. Each functional module is control autonomous, decentralizing the processor control function. By using a number of asynchronous busses to interconnect the functional modules, concurrency in the control activity is achieved. This concurrency is manifested in the overlapping of microinstruction decode and the setup of intermodule data transfers. A functional module may process a microinstruction independent of subsequent bus activity, thus allowing maximum module utilization.

This thesis is organized into three chapters which present and analyze VK-1. Chapter 2 introduces the structural concepts and system organization of VK-1, and gives detailed descriptions of the logical specifications for all system components. Chapter 3 presents and discusses system simulation studies. Chapter 4 presents details on microprogramming VK-1, and develops a large application program, namely the emulation of the IBM System/360. Chapter 5 gives conclusions of the research. Appendix material related to chapters 3 and 4 is included.

2. SYSTEM PLANNING AND ORGANIZATION

The design of the VK-1 microprocessor organization was motivated by the need for a system with certain inherent qualities which appear to be lacking in commercially available hardware. This chapter describes these design goals, and presents the functional description of the proposed hardware.

2.1 Design Objectives, and Design Overview

Microprogrammed processors are finding applications in a wide variety of environments. These range in sophistication from intelligent device controllers to central processing units. When the system designer begins to consider a system comprised of many hardware resources, the system takes the appearance of a centralized network of computing elements. The problems associated with interfacing such a collection of hardware become significant to the point of overshadowing many other design considerations. It is felt by the author that it would therefore be very desirable to have all or a significant number of the network elements based on a single structural concept. In order for this to be feasible as well as desirable, logically and economically, this basic structure must provide considerable flexibility in cost and performance, and must provide efficient data manipulations in a wide spectrum of application areas. These ideas serve as the basic design objectives for the development of the processor, called VK-1, described in this report.

As a result of the above considerations a highly modular system organization seems essential. A standard interface for all devices allows the system designer to select various functional modules for a particular application, and at the same time allows a system configuration to be modified later as required. The speed objective dictates a hardware organization capable of performing concurrent processing.

The above discussion also implies that the system be designed to allow user microprogramming. The motivations for user microprogrammed computers are discussed in references [3], [7], and [10]. In conventional systems, the usual case is that the microprogramming complexity increases as concurrency increases, thus fast systems with much operation overlap, etc. are generally difficult to program. In order to avoid this conflict and thus produce a fast, easily microprogrammed machine, the VK-1 organization decentralizes control functions. That is, the functional modules mentioned above possess a degree of autonomous control. At this point one can raise the academic question of whether or not the organization is microprogram controlled in the classical sense. The author's opinion is that it is not and the author refers the reader to a thought-provoking argument in reference 1. The industry has perpetrated misuse of the term "microprogrammed control," and so the author will refer to VK-1 as a microprogrammed machine.

As a result of the above considerations, the basic approach used in the design of VK-1 is to connect via a bus structure a collection of asynchronously operating functional modules. Concurrency of operations is allowed by having a number of autonomous busses which can operate simultaneously. The bus structure is used for the communication of both

control information and data. The control information is fed to the bus structure from the control store. This bus and control structure results in a vertically microprogrammed system, with all microinstructions taking a very simple format illustrated in figure 2.1.

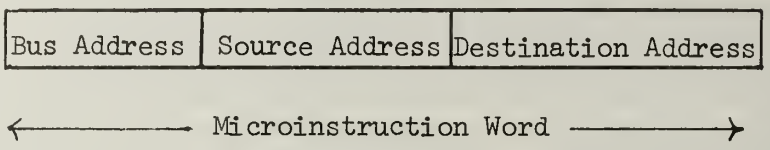


Figure 2.1 Microinstruction Format

The overall flow of information in the system is quite simple to conceptualize, when one bears in mind the microinstruction format. Figure 2.2 illustrates this flow.

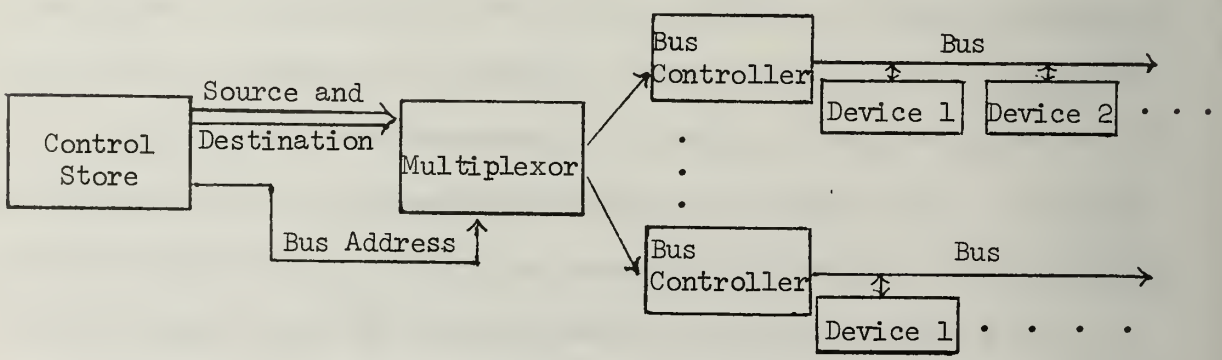


Figure 2.2 Simplified Block Diagram of VK-1

The basic approach described above affords much flexibility in system design, since the number and types of functional modules (bus devices) are variable. The number of busses may be varied from a minimum of one, and control store speed is also variable. Thus the architecture is suitable for a broad spectrum of computing power, and allows for considerable economy on the low end of the performance line.

The above discussion has attempted to give the reader a feeling for the design philosophy used in creating VK-1, and to present a brief introduction to the proposed organization. The remainder of this section gives detailed descriptions of the VK-1 hardware and principles of operation. This discussion is based on a high performance configuration. Since systems of lower capability are subsets of this hardware, this approach will enable the reader to understand the entire range of operating performance obtainable through this structure.

2.2 Multibus Organization and Bus Operations

Concurrency in VK-1 is obtained by utilizing several independent busses to perform a sequence of instructions. The result is similar to a pipelined execution unit in that the instructions are not performed completely in parallel but rather are initiated sequentially with overlapped execution. The number of busses used is variable depending on the configuration.

A bus is comprised of control lines and data lines, and is capable of performing device to device transfers over the parallel data lines. Table 2.1 lists the data and control lines comprising the bus.

Signature	No. of lines	Function
X<00-15>	16	data transfer lines
D<0-8>	9	destination address lines
S<0-5>	6	source address lines
REQ	1	cycle request
ACKS	1	acknowledge set
ACRR	1	acknowledge reset
XEN	1	transfer enable (previous instruction complete)
BCC	1	bus cycle complete

Table 2.1 Bus Signatures

Each bus has at least one bus controller associated with it which is capable of initiating a data transfer over the bus. Multiple bus controllers are present in multiprocessor systems and when sophisticated autonomous bus devices are used. These will be discussed later. Bus transfers are fully asynchronous and interlocked. All bus devices require logic for address recognition and handshake protocol.

A bus cycle is comprised of the following events. First, of course, Source and Destination fields of the next microinstruction are multiplexed (via the bus address field) to the appropriate bus controller. The bus controller accepts this information and signals the multiplexor as soon as it recognizes that the bus is available. The bus controller accepts the source/destination field, and asserts them on the S and D lines. The source device recognizes its address and as soon as it's ready, asserts the data to be transferred onto the X lines, and sets the ACK

flipflop thru ACKS. The destination device recognizes its address, waits if necessary, until it is ready, waits if necessary until ACK is set and XEN is true, and then loads the data on the X lines into its internal register. As it does this, it signals the bus controller and source that the transfer has completed by resetting the acknowledge flip-flop.

As was described, several conditions can cause a bus cycle to be suspended pending the completion of some other operation. These conditions will be further described. The timing and flow diagrams of figures 2.3 and 2.4 should clarify this operation.

The number of bus devices is variable within the limits of the address space of the source address and destination address fields. The size of the microinstruction is also variable so that for a given system, the maximum number of desired source and destination devices will first be determined, then the microinstruction length fixed accordingly. On the order of ten devices per bus seems quite adequate for systems considered in this research.

Bus devices may be assigned to busses in a variety of ways. A device which may function as both a source and a destination (e.g., a bus register) may occupy a source address on one bus and a destination address on a different bus. Also a device may occupy source addresses or destination addresses on two or more busses. The fact that a device may suspend a bus cycle if it is "not ready" allows for this type of bus assignment. Of course the complexity of the device controller increases when multiple address assignments are made.

2.3 Instruction Stream Generation and Distribution

As indicated in previous discussion, a stream of microinstruction source/destination fields are multiplexed to the various bus controllers

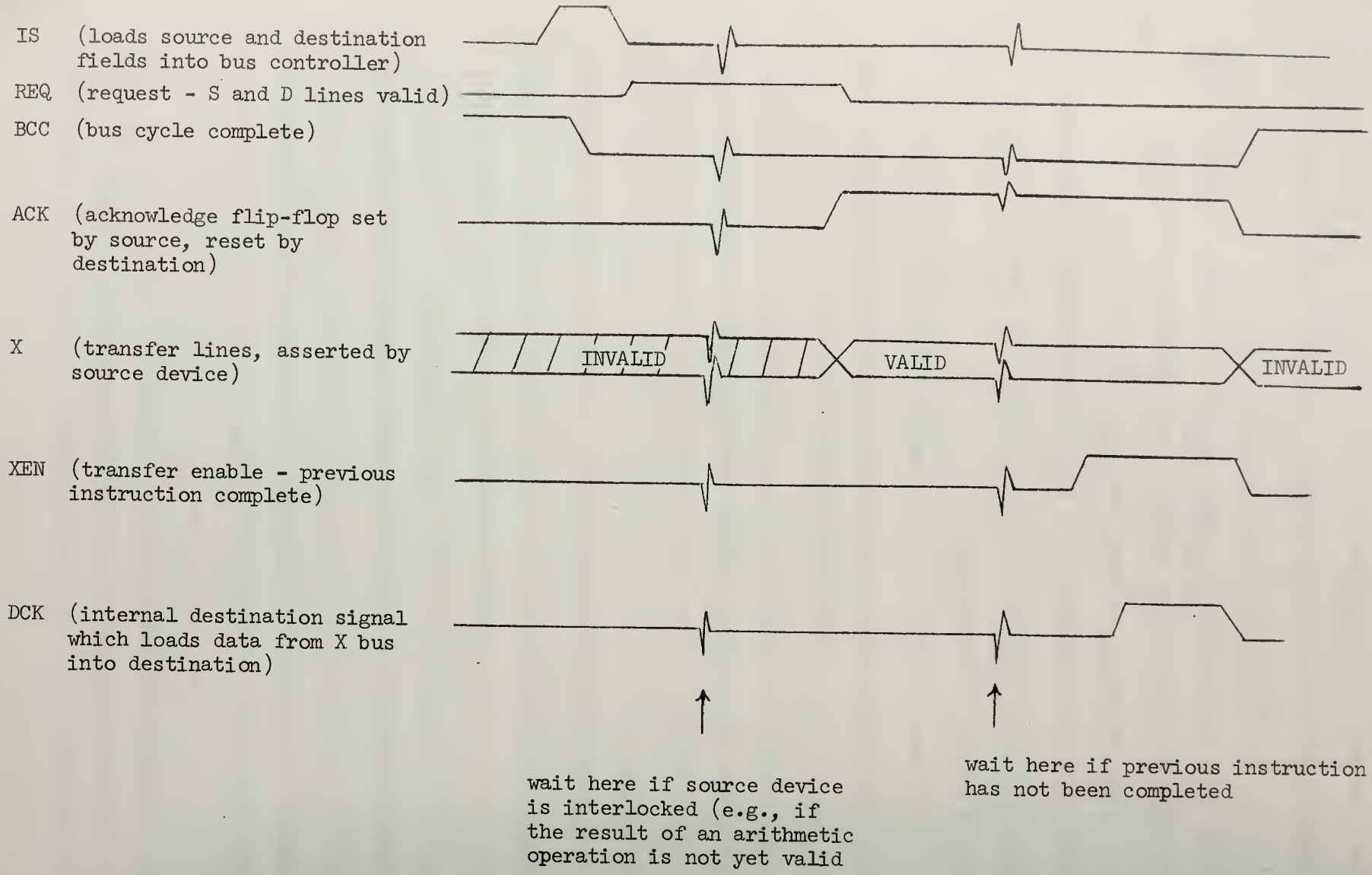

IS (loads source and destination fields into bus controller)
REQ (request - S and D lines valid)
BCC (bus cycle complete)
ACK (acknowledge flip-flop set by source, reset by destination)
X (transfer lines, asserted by source device)
INVALID
VALID
INVALID
XEN (transfer enable - previous instruction complete)
DCK (internal destination signal which loads data from X bus into destination)
wait here if source device is interlocked (e.g., if the result of an arithmetic operation is not yet valid
wait here if previous instruction has not been completed

Figure 2.3 Bus Cycle Timing Diagram

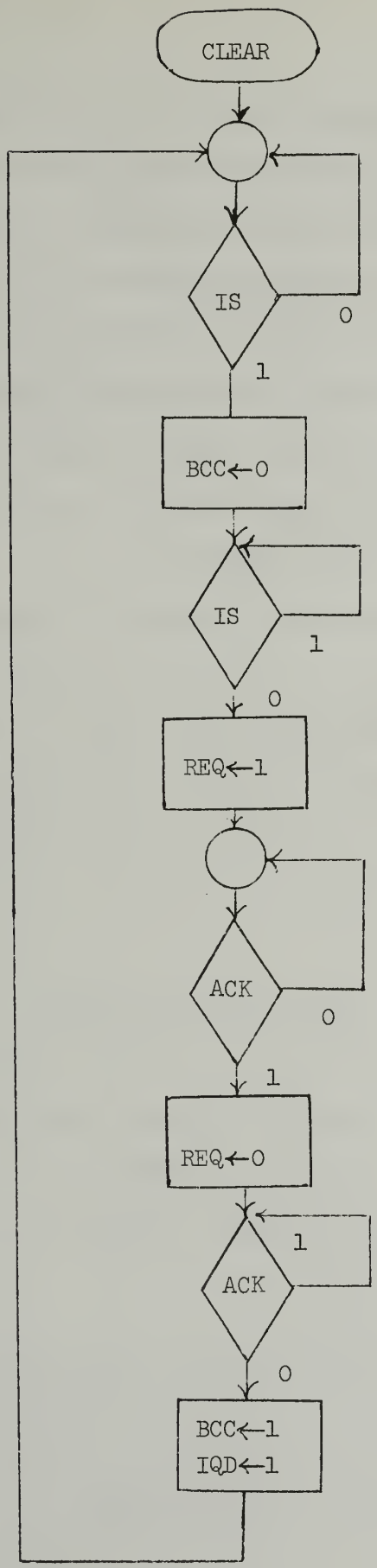


Figure 2.4 Bus Controller Flow Diagram

via the microinstruction bus address fields. In the discussion of bus cycles, it was pointed out that a bus cycle could not be completed until the previous bus cycle was completed. This is required to assure sequential execution of the microprogram. It was also mentioned that a bus controller cannot accept a source/destination word when that bus is in the process of executing a bus cycle. For maximum microinstruction throughput, we would like to initiate each bus cycle at the earliest possible moment. These ideas form the basic constraints on the definition of the control store and multiplexor subsystem illustrated in figure 2.2.

In order to match control store bandwidth to execution rate, a very fast memory system is required. To achieve this bandwidth, the control store is comprised of four interleaved memory modules. Each module has its own address register and controller. Output data from the memory modules is not buffered. Four control signals are associated with each memory module, and are defined in table 2.2.

Signature	Definition, Description
MAVi	module available - module i may be accessed
DAVi	data available - module i data output is valid
CIi	cycle initiate - begin memory cycle in module i
AIi	address increment - increment address register i

Table 2.2 Control Store Module Control Signals

Control store modules are accessed according to the following procedure. The two least significant address bits of the control store address are decoded to select one of the four modules. If that module is available (MAVi = 1) then a cycle is initiated in that module (CIi is pulsed). This control procedure is illustrated by the flowchart of figure 2.5. (Note in figure 2.5 and other flowcharts which follow that some control signals are levels and some are pulses, as indicated.)

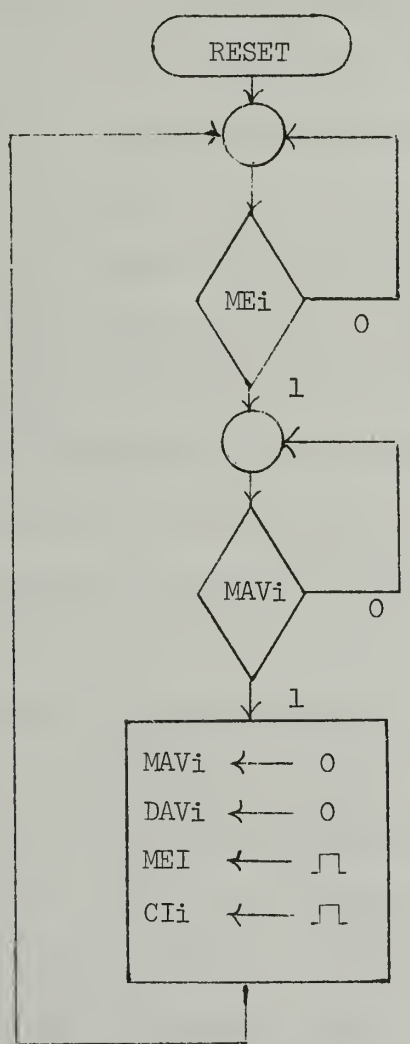


Figure 2.5 Cycle-Initiate Controller for Control Store Module i

The initiation of a memory cycle in a particular module also starts a timer (one-shot) which causes data available (DAVi) for that module to become true after a time delay equal to the access time of the memory module. This operation assures that the control store modules will be accessed as soon as possible, maximizing microprogram look-ahead. The module available signals (MAVi) which are reset when module i is initiated are set by the bus controllers which receive the microinstructions. Thus these controllers are interlocked to the cycle initiate controllers, so that a memory access cannot be initiated in a module before the current output data (microinstruction) has been received by a bus controller.

Parallel data paths connect the output data lines of the control store modules to the microinstruction multiplexor. Figure 2.6 illustrates the configuration under discussion. A modulo four counter, the active module counter, is used to indicate to the multiplexor which of the control store module output lines contain the next microinstruction to be transferred to a bus. When data available (DAVi, where i = [AM]) for the active module becomes true, then the output lines of that memory module are switched onto the I bus and the microinstruction bus address field is decoded by the bus controllers. The controller of the addressed bus then waits if necessary for bus-cycle-complete (BCC) to be true, then strobes the source and destination fields of the microinstructions into the bus instruction register. When this instruction strobe occurs, the active module counter is incremented so that it now points to the next (modulo 4) module. Also module available (MAVi) for the module just unloaded is set, and the address register is incremented. This instruction strobe initiates the bus cycle corresponding to this microinstruction as explained earlier. A parallel

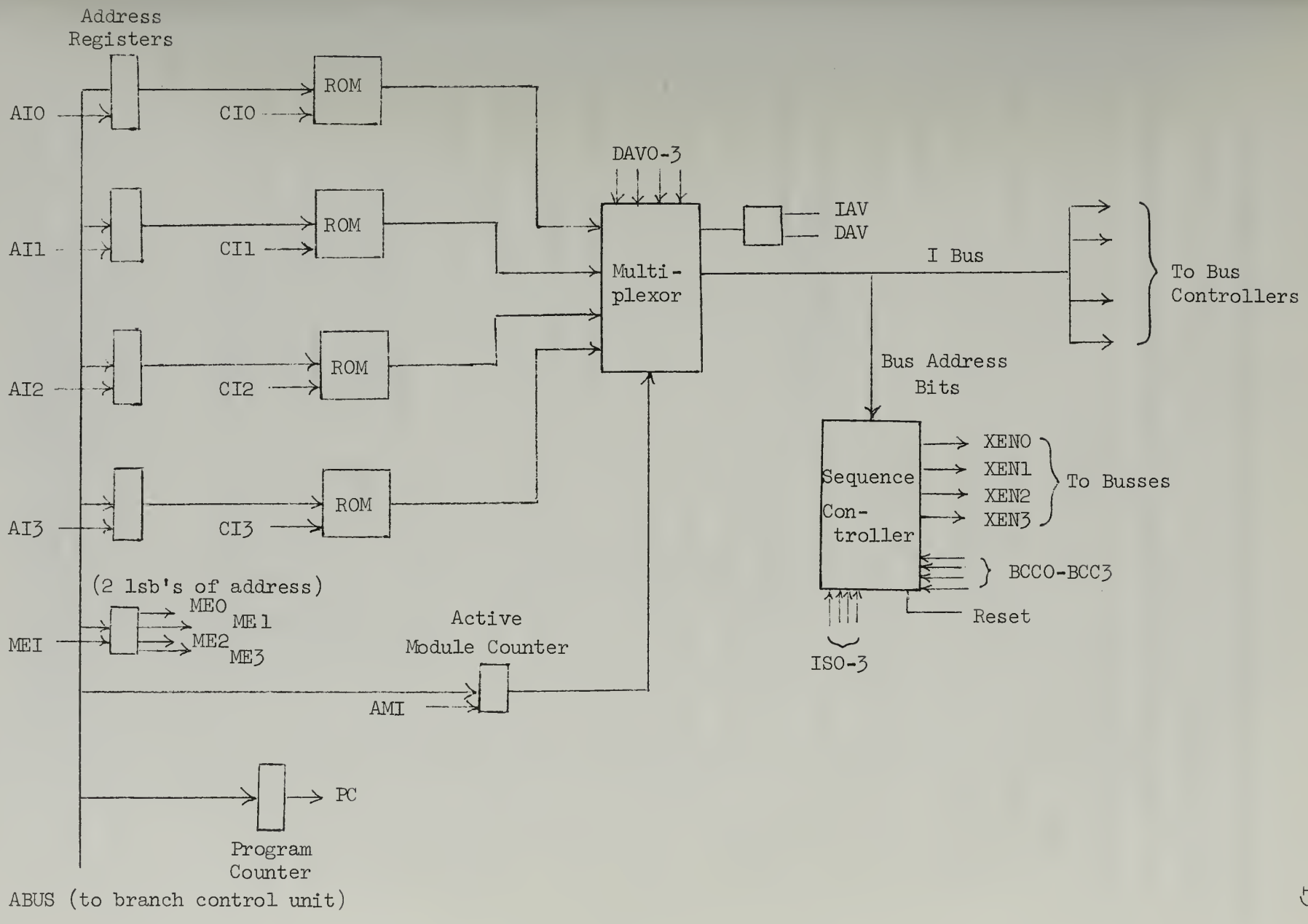


Figure 2.6 Instruction Stream Generator

activity is taking place in the sequence controller and this will be described shortly.

The actions of the instruction strobe controller (1 per bus) are summarized in figure 2.7.

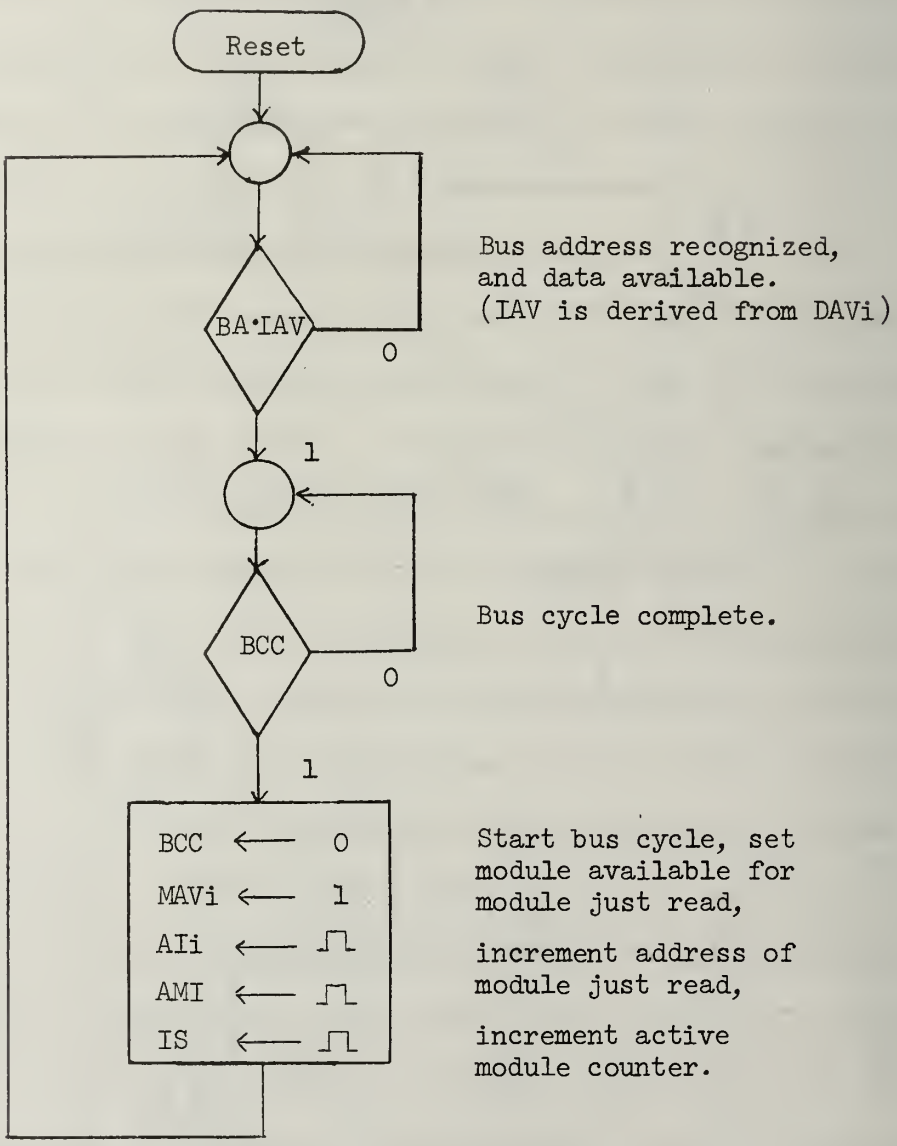


Figure 2.7 Instruction Strobe Controller

The reader should be able to understand the basic functions of the VK-1 instruction sequencing mechanism at this point. Note that if a bus cycle becomes suspended by a bus device interlock, as many as 3 subsequent instructions can be started (one on each of the other three busses), and memory accesses for the next 4 instructions beyond the 4 already strobed into bus control registers can be started and completed. Thus lookahead can proceed to up to seven microinstructions beyond the instruction which is interlocked.

In cases where more than one instruction has been strobed into its bus controller (which is the normal situation) it is necessary to remember the order in which these instructions were strobed if sequential execution is to be maintained. This function is handled by the sequence controller shown in figure 2.6. This mechanism consists of a queue structure. Whenever an instruction is strobed by a bus controller, the bus address bits for that instruction are loaded into the rear of the queue. Whenever a bus cycle is completed, the bus address at the front of the queue is discarded and the front of the queue becomes the bus address of the instruction strobed just after the instruction which has just completed. In order for a bus cycle to complete, the address of that bus must be at the front of the queue. Thus the sequence controller generates the four transfer enable (XEN0-3) signals, one for each bus. The XEN signal which is true at any time corresponds to bus whose address is at the front of the queue. Figure 2.8 illustrates a valid sequence of states for the instruction queue.

Obviously the queue can hold a maximum number of bus addresses equal to the number of busses (4 in the machine being discussed). Queue overflow can not occur because an instruction cannot be strobed by a bus controller

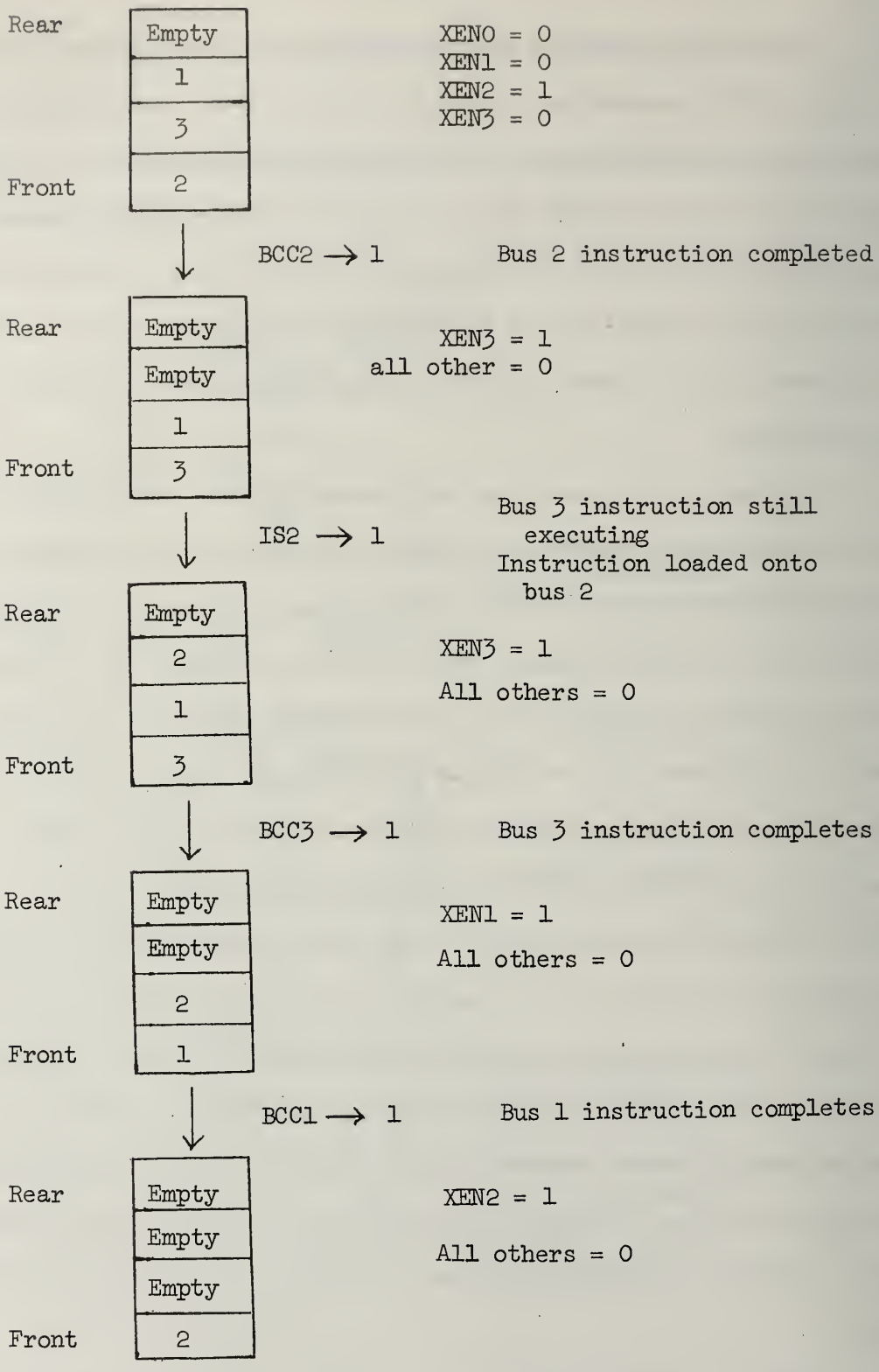
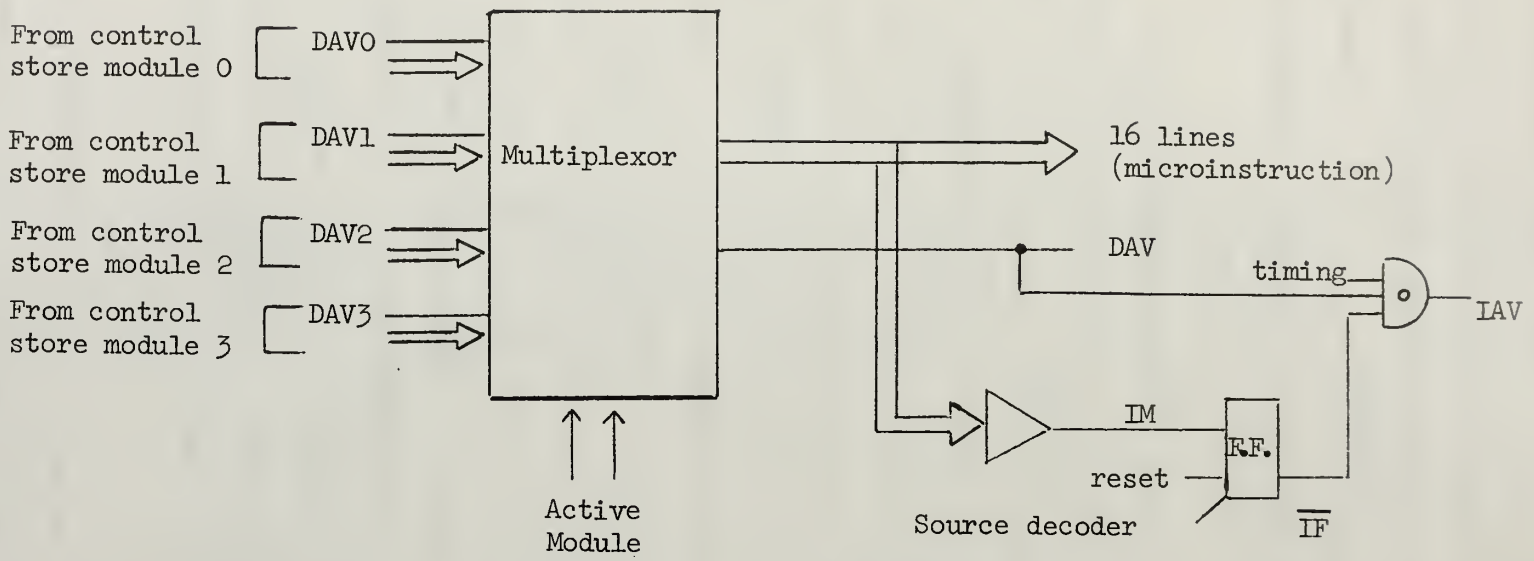


Figure 2.8 A Typical Sequence of Activity in the Sequence Controller

if that bus is busy, and if the bus is free there is at least one space in the queue. The hardware required to implement the sequence controller is straight forward consisting of two 4 bit shift registers (one for each bit of bus address) and a decoder which activates the appropriate XEN line. The logic which loads the queue is driven by the 4 instruction strobe (IS) signals, and the logic which unloads the queue is driven by the four bus cycle complete (BCC) signals.

From the standpoint of efficiency of microprogramming, it is essential to be able to obtain constants from control memory in a literal fashion. That is by providing a double word microinstruction format where the first word is the usual bus address, source address, destination address format, and the second word is the data to be used as the source data. In this case, the source field of the first word indicates that the data source is immediate (i.e., from control store). To implement this feature, each bus is provided with an immediate data register (IDR) which may be specified as the source. This register will always be loaded with the next word accessed from control store when it is specified as the source. The source address corresponding to IDR must therefore be recognized by the bus controller. When this happens a flag is set which loads the next word appearing on the I-bus into the IDR. The data available (DAVi) signals coming from the memory modules are changed to instruction available (IAV) for a normal instruction and to data available (DAV) when the word being multiplexed is a data word. Figure 2.9 depicts the logic involved in this hardware.

This section has described logically the internal operation of the instruction generation segment of the VK-1 hardware. As was mentioned at the beginning of this discussion, a high performance model is the topic of



IM - immediate data instruction detected

IF - immediate data flag

DAV - data available

IAV - instruction available

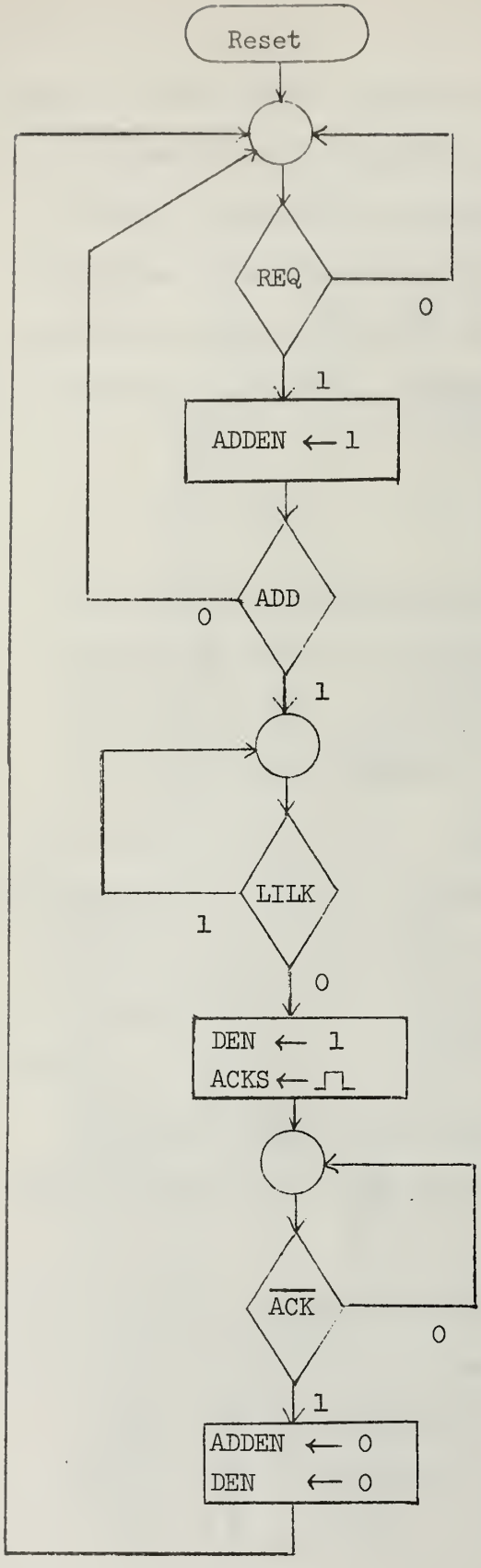
Figure 2.9 Immediate Data Logic

discussion. From the above description it should be evident that a single control store module could be used if less throughput were allowable. Also the modifications required to facilitate more or fewer than 4 busses is evident. Further discussion of some of the hardware described above will be required when bus devices are discussed. The most notable of these is involved with branching which alters the control store effective address and the contents of the completion queue.

2.4 Bus Devices and Instruction Types

The previous discussion described the mechanism which produces source-destination type instructions, and loads them into the addressed bus controller. Section 2.2 defined bus operation. This section will discuss the nature of the bus devices which respond to the source and destination addresses propogated on the bus. One of the basic design concepts of the VK-1 is that bus device assignment is variable in configuration and device type, and assignment is done to optimize processor efficiency for a particular application. This section will describe the design of several bus devices. The designs chosen for inclusion in this report are those which will typically be required regardless of application area. Also described are those devices which were selected for the IBM System/360 emulator described in chapter 4.

Before proceeding with the descriptions of specific devices the bus interface controllers for source devices and destinaton devices will be described. The requirements for this logic are apparent from the bus description previously presented. Figures 2.10 and 2.11 show the flow diagrams for these controllers.



Wait for REQ (request) from bus controller

Examine source address lines

Address recognized?

Yes

Check for local interlocks*

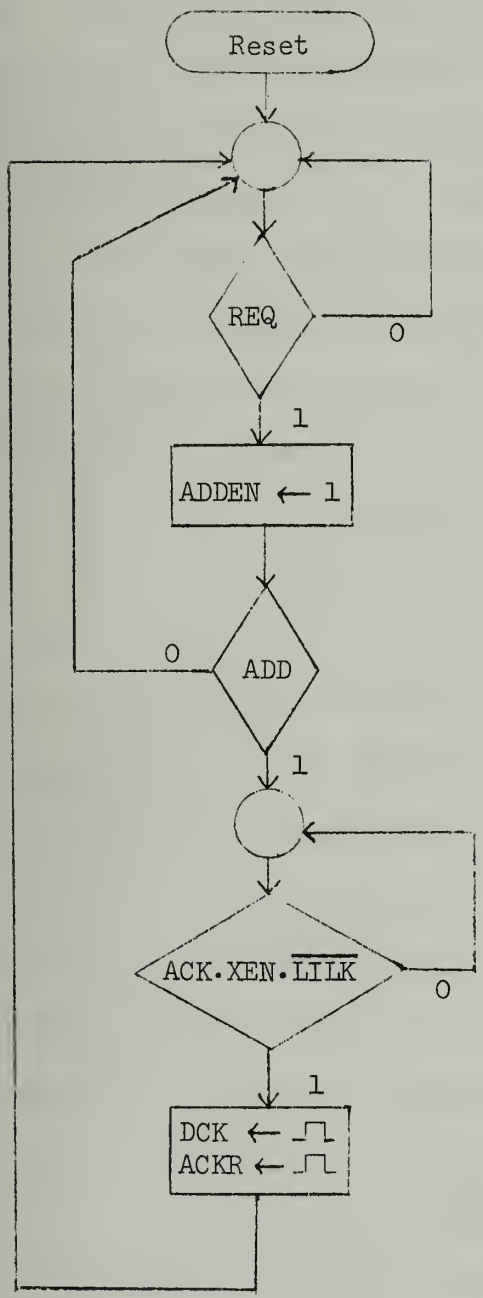
Enable data onto X lines and acknowledge

Wait for destination to clear acknowledge

Disable data and decode

*Check for local interlocks (These depend on device type)

Figure 2.10 Source Bus Controller



Wait for REQ (request) from bus controller.

Examine destination address lines.

Addressed recognized?

Yes.

Wait for ACK (X lines valid)
XEN (from seq cont)
$\overline{LILK}$ (clear of local interlock)

Load data from X bus and clear ACK (acknowledge).

Figure 2.11 Destination Bus Controller

It is apparent that the microinstruction repertoire is determined by the set of bus devices which are selected for a particular system. Many of the devices are actually functional units which may perform a number of operations. Obviously such devices require control information. Examples are read-write memory and arithmetic units. Such command information is imbedded in the source or destination address fields. Thus one may consider each function to possess a bus address giving some physical devices several addresses. The author has found it more convenient to consider the address field to possess control fields as well as device addresses. This approach will be utilized in discussing various bus devices.

2.4.1 Arithmetic and Logical Unit

The design specifications chosen for the arithmetic and logical unit are based on the goals of speed and versatility. Most emulation applications require a significant amount of bit manipulations so that powerful logical instructions are of considerable value. Implementation of a powerful set of logical operations is quite feasible with currently available MSI devices. Provisions for convenient multiple precision arithmetic operations should also be made, allowing efficient manipulations of a variety of target data formats. Also for convenience and efficiency, it is worthwhile to consider monitoring a number of arithmetic and logical conditions.

Evaluation of these considerations has led to design with the structure shown in figure 2.12.

In figure 2.12 notice that the inputs and outputs may reside on different busses, and in general will so that loading of the 2 operands may be done concurrently. Also the function command information for the ALU is

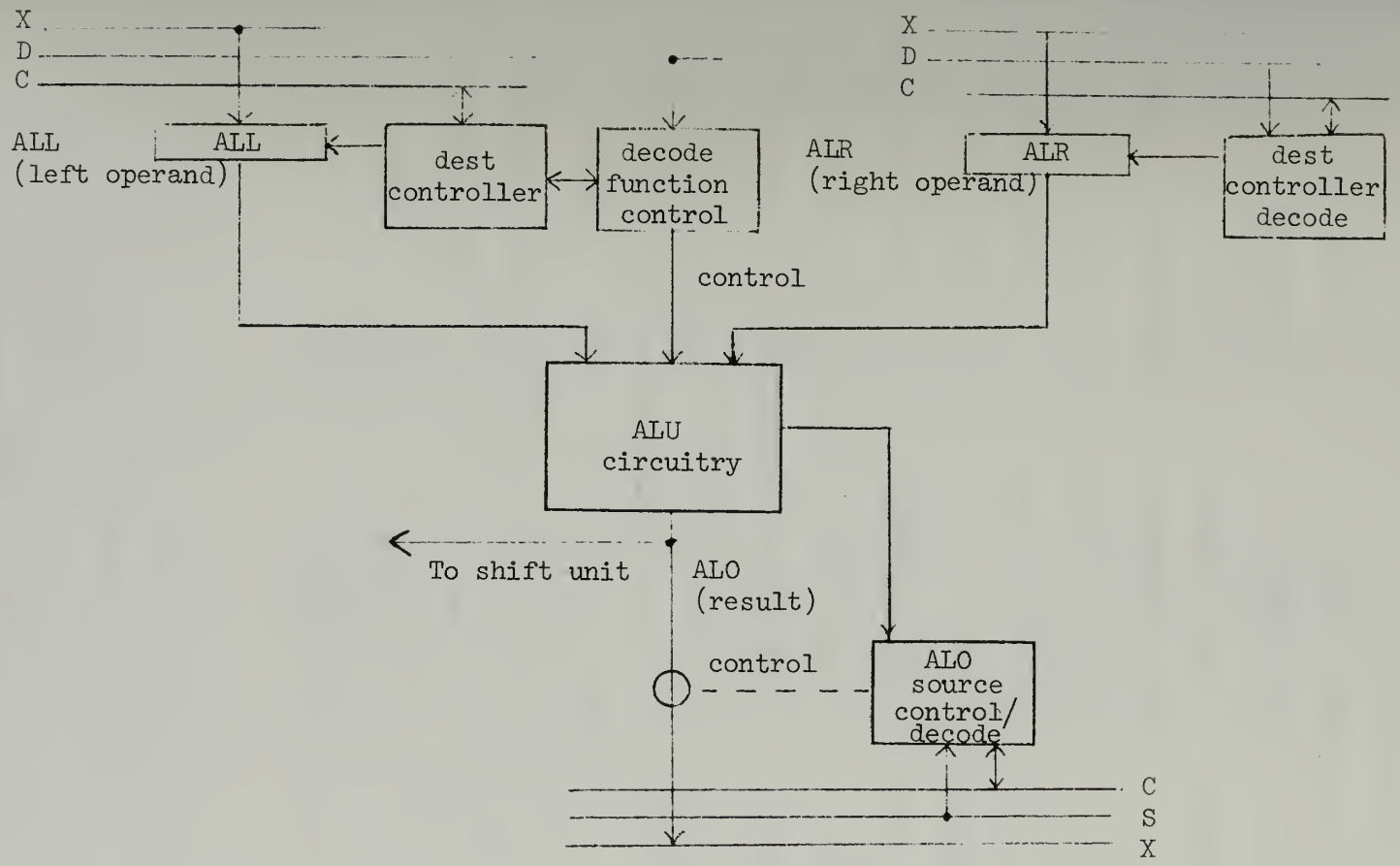


Figure 2.12 Arithmetic and Logic Unit

provided when the ALL is loaded. Loading of the ALL causes the ALU to begin its cycle, and thus the ALR must contain the proper argument before that time. During an ALU cycle, the inputs and outputs will become internally interlocked so that if for example the ALO is addressed before the ALU cycle has completed, the source will interlock, i.e., not complete that bus cycle until the ALU has completed and the ALO contains valid data. Also addressing an operand input will interlock in the same way if the previous ALU cycle is still in progress. Table 2.3 lists the various operations which could be considered. In applications which do not require such an extensive set it is desirable to decrease the number of allowable operations thus saving destination address space and reducing hardware costs. The set illustrated is probably a super-set of those operations required for most applications.

The ALU automatically generates and saves condition flags during each ALU operation. Table 2.4 describes these flags. Condition flags are saved in the branch status register for examination by the branch control unit to be discussed later in this chapter.

Multiple-precision operations are facilitated by the ADDC and SUBC instructions which utilize the previous carry as input. These instructions also preserve valid condition codes throughout a multiple-precision add or subtract since they can only clear the Z bit.

2.4.2 Shift Unit

Shift operations are rather frequent in emulation because a variety of word formats must be decoded, and manipulation of arbitrary fields must be done. Thus a reasonably powerful shifter is of considerable value. Since many shift operations are preceded by ALU operations such as mask types, the shifter should have rapid access to the ALU result. It is also

Hexadecimal Function Code	Mnemonic		
00	CLR	clear ALO to all zeros	ALO ← 0...0
01	SET	set ALO to all ones	ALO ← 11...1
02	TRL	transmit left	ALO ← ALL
03	TRR	transmit right	ALO ← ALR
04	NOTL	NOT left	ALO ← $\overline{\text{ALL}}$
05	NOTR	NOT right	ALO ← $\overline{\text{ALR}}$
06	AND	left AND right	ALO ← ALL·ALR
07	OR	left OR right	ALO ← ALL+ALR
08	NAND	left NAND right	ALO ← $\overline{\text{ALL·ALR}}$
09	NOR	left NOR right	ALO ← $\overline{\text{ALL+ALR}}$
0A	XOR	left exclusive OR right	ALO ← ALL⊕ALR
0B	MASKR	mask right	ALO ← $\overline{\text{ALL}}$·ALR
0C	MASKL	mask left	ALO ← ALL·$\overline{\text{ALR}}$
0D	INSR	insert right	ALO ← $\overline{\text{ALL}}$+ALR
0E	INSL	insert left	ALO ← ALL+$\overline{\text{ALR}}$
0F	EQU	left EQU right	ALO ← $\overline{\text{ALL}}$·$\overline{\text{ALR}}$+ALL·ALR
10	NEG	negate left, 2's complement	
12	ADD1	add, 1's complement	
13	ADD2	add, 2's complement	
14	ADDC	add, with previous carry	
15	SUB2	subtract, 2's complement	
16	SUBC	subtract, 1's complement with previous carry	
17	DEC	decrement left	
18	INCL	increment left	
19	INCR	increment right	

Table 2.3 ALU Operation Set

Mnemonic	Description
OV	Overflow. Set by arithmetic operations when the result overflows the 16 bit result register. OV is set if $L_0 R_0 \overline{\emptyset}_0 + \overline{L}_0 \overline{R}_0 \emptyset_0$ and cleared otherwise (L_0, R_0, $\emptyset_0$ represent the most significant (sign) bit of the ALL, ALR, ALO registers respectively.)
C	Carry. Set by arithmetic operations. C is set if a carry out of the most significant bit occurs and is cleared otherwise.
Z	Zero. Set by logical and arithmetic operations. Z is set if all bits of the ALO are zero, and cleared otherwise. ADDC and SUBC can only clear the Z bit.
N	Negative. Set by arithmetic and logical operations. Set if the most significant (sign) bit of the ALO is one, and cleared otherwise.

Table 2.4 ALU Condition Flags

desirable to allow the shifter rapid access to its previous result so that multiple shift operations are conveniently executed.

These objectives may be accomplished by allowing the shift unit (SHR) to accept a source from the X lines as usual, and by allowing it to also accept a "pseudo" source internally from itself or the ALO. When a pseudo source is specified, the SHR responds to both source and destination lines and indicates to the bus controller that a bus cycle has been completed (by setting and resetting the ACK flip-flop). This arrangement gives the SHR rapid access to the ALO and SHR even if the SHR input is on a different bus than the ALO and the SHR output. Figure 2.13a illustrates the SHR.

The shift control command is contained in the destination field. This command format is illustrated in figure 2.13b. With the commands shown it is possible to execute an arbitrary shift in two microinstructions.

2.4.3 Branch Control Unit

The branch control unit (BCU) is the mechanism which allows conditional alteration of the sequential execution of microinstructions. This unit must possess considerable capability because of the look-ahead which is normally taking place in a given instruction stream. Since the instruction stream generator is pipelined, it is desirable to accomplish branching with minimum loss of speed-up in the pipeline. That is, it is desirable to execute a branch without having to flush, then refill the pipe. The BCU to be described has this capability when the microcode will allow branch look-ahead. It is also desirable to have the capability to conditionally branch on a variety of conditions, and to obtain the branch address from a variety of possible sources. These design goals motivate the specifications for the BCU.

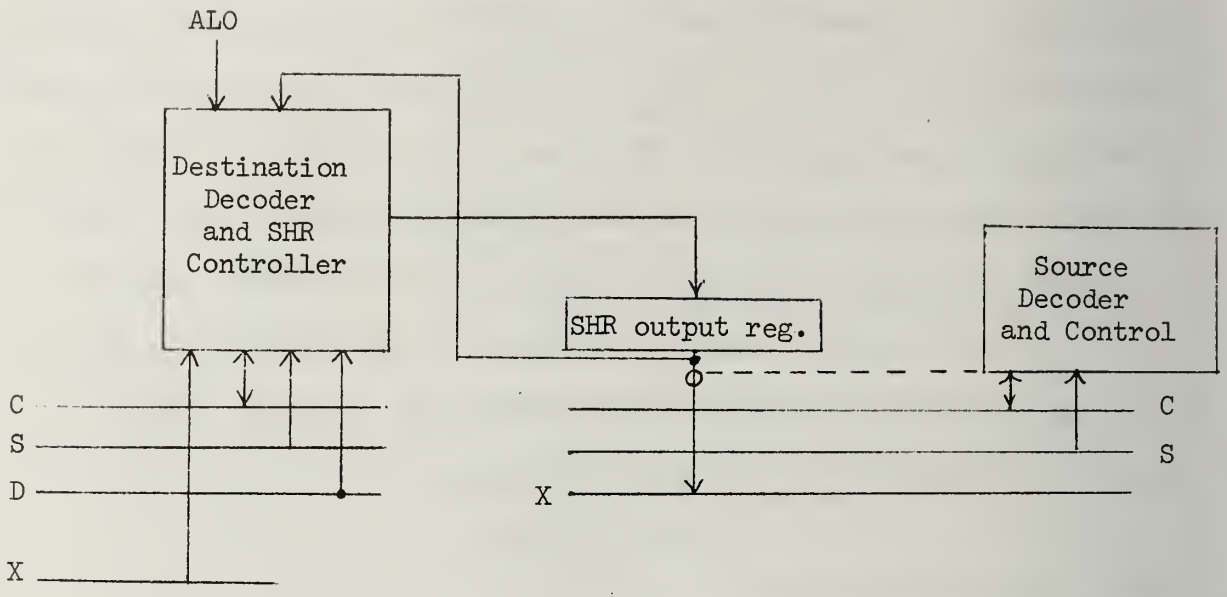


Figure 2.13a Shifter Unit Block Diagram

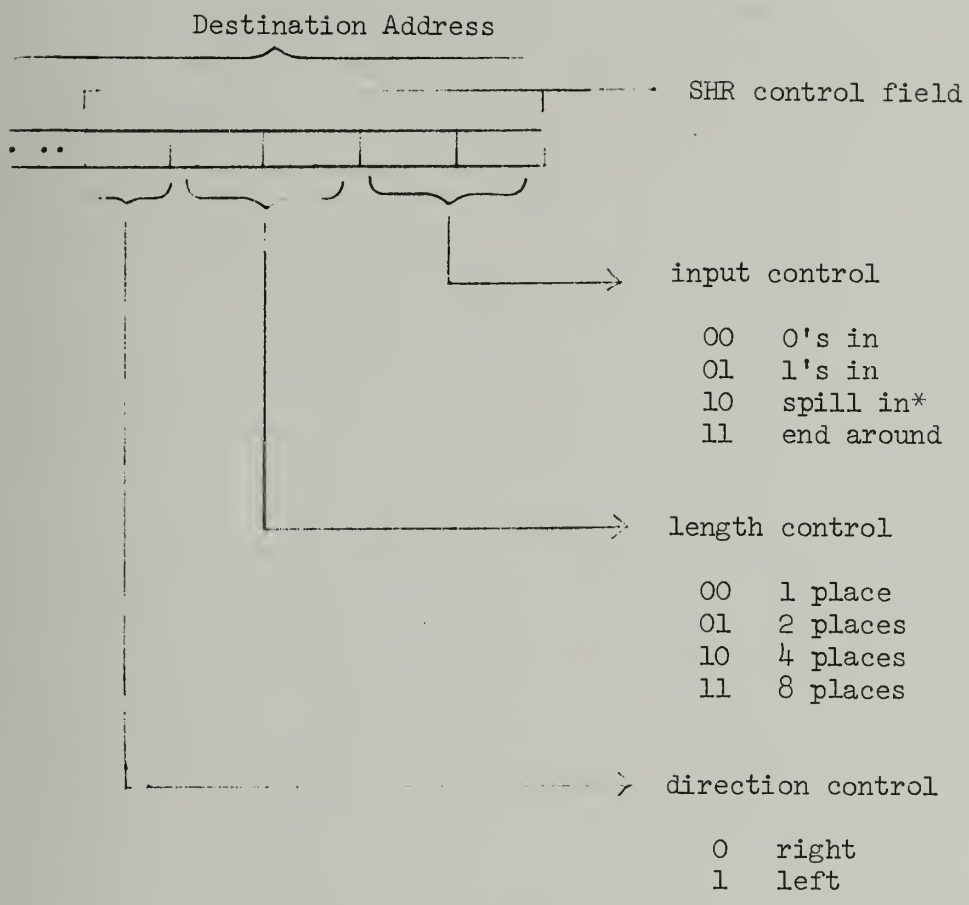


Figure 2.13b Shifter Unit Control Format

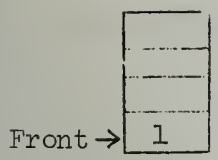
Branching is accomplished by accessing the BCU as a destination. The source specified contains the branch address which is loaded into the microinstruction counter if the branch is successful. If the branch is unsuccessful, execution proceeds without altering the microinstruction counter. When a successful branch occurs, the source word is loaded into the program counter, the 2 least significant bits are used to initialize the module enable register and the active module counter, and the remaining bits (all except the 2 lsb's) are loaded into the four address registers. (Refer to figure 2.6.) Before this action can occur, however, the sequence controller must be examined, and properly initialized.

The term "branch look-ahead" as used in this discussion refers to instances when a branch instruction is preceded by instructions which do not affect the branch condition or the branch address. In this situation VK-1 can effect very efficient branching. These instructions, called post-instructions, are stored in control store at sequential addresses following the branch instructions. The branch instruction contains a field designating the number of post-instructions which follow the branch. Any number of post-instructions up to N-1, where N is the number of busses may be specified. However, the post-instructions must specify distinct busses, and may not specify the bus on which the branch is being executed. The BCU waits until all post instructions are loaded into bus control registers before strobing the branch address into the microinstruction counter. The effect of this is to allow the access of the instruction at the branch address to be taking place while the post-instructions are executing. In the case where zero post-instructions are specified, the flushing and refilling of the pipe must take place. The queue contained in the sequence controller as described in section 2.3 facilitates this activity.

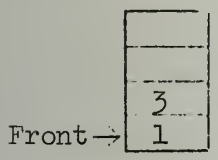
An example will clarify this discussion. This example illustrates the execution of a branch with 3 post-instructions (on a four bus machine). The sequence of events to be described begins when the branch instruction reaches the front of the instruction queue. Figure 2.14a gives a section of code. Figure 2.14b shows the sequence of states which the instruction queue experiences during the execution of this code. Recall that the bus whose address is at the front of the queue is enabled to complete, a bus address is entered into the rear of the queue when that bus control register is loaded with a microinstruction, and the queue is "popped" when an instruction completes. Note in figure 2.14 that no delay was experienced because of the branch. Such coding is therefore optimal. Code such as this can often be generated if the programmer is aware of the advantages. Since the post-instructions are issued, but not necessarily completed before the instruction counter is altered, they must not alter the branch condition or branch address. The post-instructions are completed before the branch becomes effective, however, since they are ahead of the instruction from the branch address in the instruction queue. Notice that all that is required to implement this plan is logic which detects the number of bus addresses in the queue (or equivalently the position of the "rear" queue pointer). The following situation must also be allowed for. If, when the branch instruction reaches the front of the instruction queue, the queue length exceeds M+1 (M is the number of post-instructions), then too many instructions have been issued (assuming the branch is successful). In this case, the queue must be initialized to the length which includes only M+1 elements. Bus addresses in the queue which reside behind the M+1 position represent instructions which have been issued but which should not be executed. Bus resets must be issued on these busses to abort execution of

Control Store Address		Bus Address	Source Address	Destination Address	
	L	1	S_B	BCU	with M=3 post-instruction
post-instructions	L+1	3	S_1	D_1	
	L+2	0	S_2	D_2	
	L+3	2	S_3	D_3	
	⋮				
	$[S_B]$	1	S_a	D_a	
	$[S_B]$ +1	0	S_b	D_b	
	⋮				

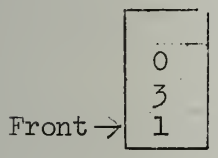
Figure 2.14a Example Code for Branch



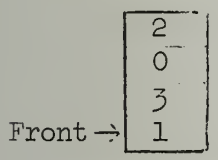
Branch instruction has been issued on bus 1, and is waiting for the three post-instructions to be issued.



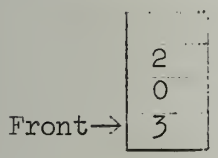
Post-instruction on bus 3 issued.



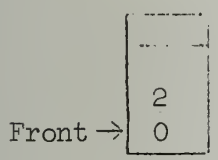
Post-instruction on bus 0 issued.



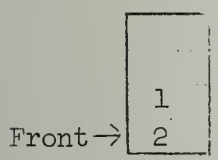
Post-instruction on bus 2 issued. Since queue contains M+1 entry's, all post-instructions have been issued.



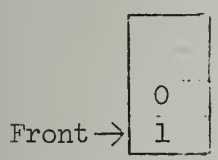
BCU has strobed $[S_B]$ into micro-instruction counter and completed the bus-1 cycle. The access of $[[S_B]]$ from control store is now in progress.



Post-instruction, bus 3 has completed.



Post-instruction, bus 0 has completed, and $[[S_B]]$ has been read from control store.



Post-instruction, bus 2 has completed and the instruction at the branch address is at the front of the queue. The next instruction has also been issued.

Figure 2.14b Instruction Queue of Sequence Controller During Execution of Branch with Post-instructions

these bad instructions. This is accomplished by reseting the acknowledge flip-flop in appropriate bus controllers. (The destination involved is aware of the abort because the acknowledge has been reset by someone else while he is waiting for transfer enable (XEN_i).)

Notice also that in the described design, the source of the branch address may be any valid source device. This allows extreme flexibility in programming. Examples are convenient branch table decoding and subroutine linkage.

Figure 2.15 diagrams the BCU. The portions of the BCU remaining to be discussed are the Branch-Status Register (BSR), and the decision logic. The purpose of the BSR is to remember various machine conditions so that they may be tested by the BCU. The condition codes generated by the ALU are stored in the BSR. The BSR may also contain flag bits which may be set and cleared by other devices. Further capability may be added to the BCU by including a branch test register (BTR) which may be a source or destination. If this is done, then BCU commands can specify testing of either the BSR or BTR. This addition allows the BCU to perform conditional branches on any bit of any word which can be transferred into the BTR. (This much capability is not normally required and is not assumed in the firmware discussed in Chapter 4.) Figure 2.16 shows the BCU command format (which is inherent in the destination address). There does not appear to be a simple way to test for boolean combinations of bits in the BSR or BTR.

2.4.4 Inter-Bus Communication

An obvious problem which is inherent in the multibus structure of VK-1 is the communication of data between busses. It is not economical, in general, to configure all devices so that they are each accessible from

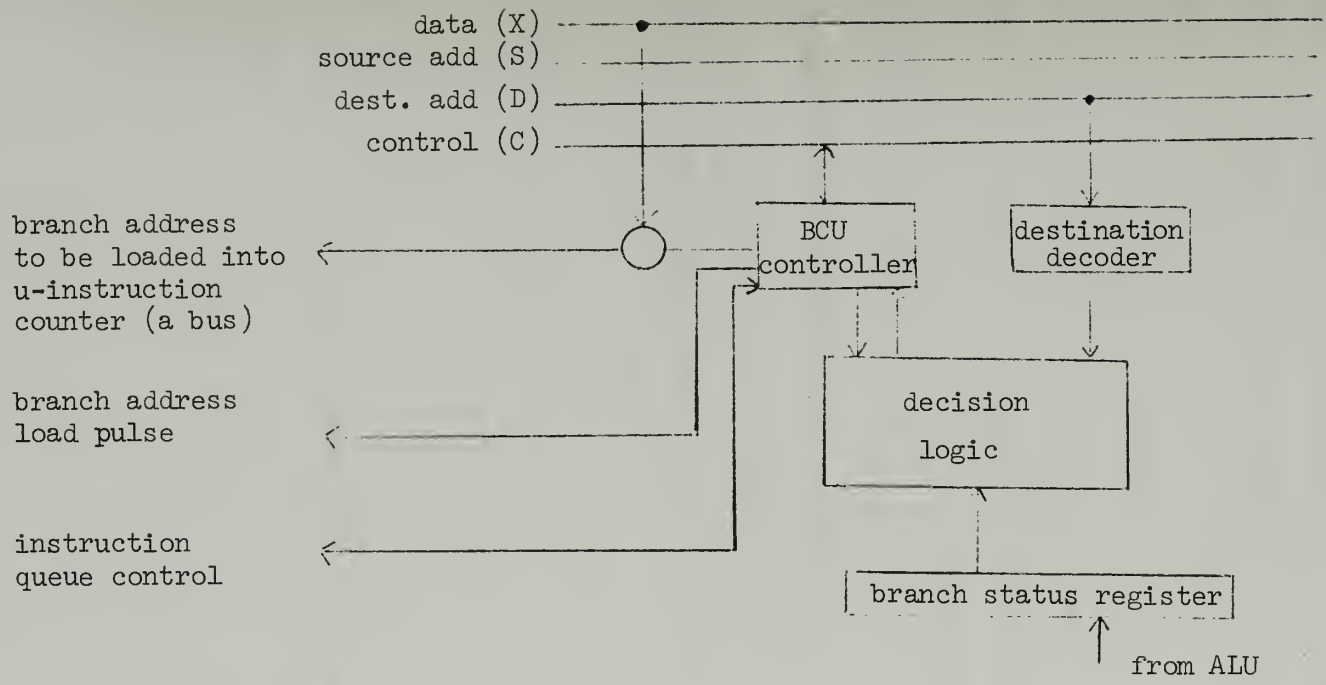


Figure 2.15 Branch Control Unit

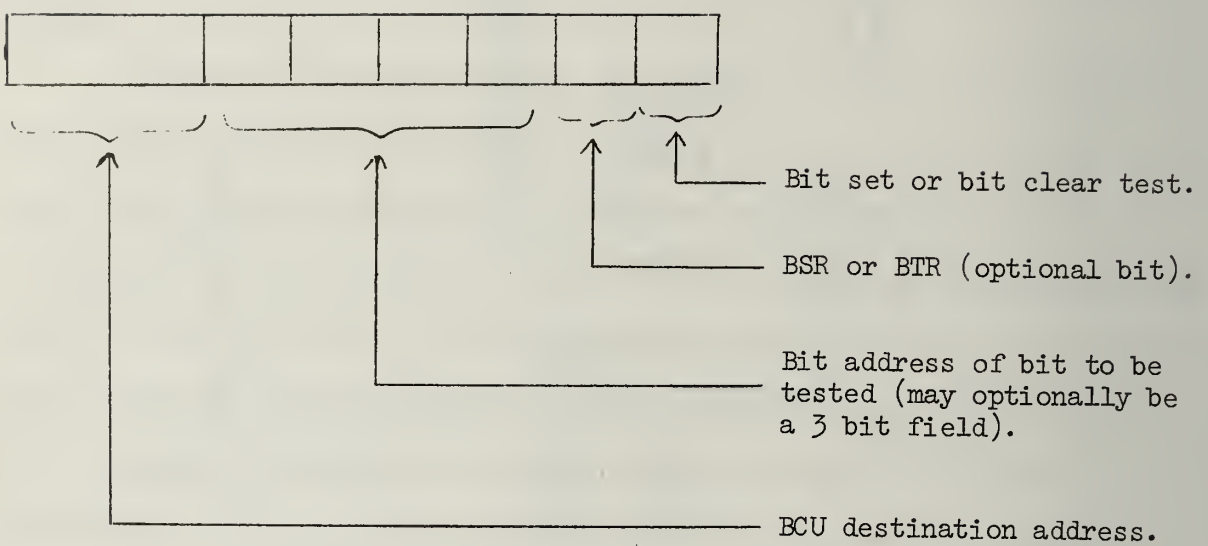

Bit set or bit clear test.
BSR or BTR (optional bit).
Bit address of bit to be tested (may optionally be a 3 bit field).
BCU destination address.

Figure 2.16 BCU Command Format

all busses. As a result, it is necessary to consider ways of effecting data transfer from a source on one bus to a destination on another. A straight-forward solution to this problem is to provide a register called a Universal Communications Register (UCR) which may be loaded from a source on bus A and may then, on a subsequent microinstruction, load its content into a destination on bus B. If it is expected that such transfers will occur frequently it may be desirable to supply several such devices so that delays caused by the UCR being busy do not accumulate. However, if such transfers are frequent it is probably desirable to first reconsider the bus assignments of the various devices. Since bus devices which act as sources and destinations may be connected across two busses, it should be possible to minimize the need to do interbus transfers. Consequently strong connectivity of busses by UCR's should not be generally required. This is desirable because two bus cycles which are not highly "overlappable" are required to accomplish the transfer.

2.4.5 Scratch-Pad Memory

In order for a processor to achieve high throughput, it is necessary to have frequently used data rapidly accessible. This is normally done with high-speed register type memory which is commonly referred to as scratch-pad memory. In preparing microcode it has been observed that accessing modes in addition to random access would be useful, and would save a number of microsteps. A simple example of such a case occurs when a 32 bit target machine is to be emulated on a 16 bit host. One would expect in this case, that many of the references made to scratch memory will first access one word, and then access the next sequential word. In this example, a microstep could be saved for each of this type access made if the scratch-pad could respond to an "increment/read" operation (loading of

the address register for the second read is eliminated). This idea could be further extended so that two microsteps could be saved on each of this type of reference. This could be accomplished by executing an increment-read which is issued when the first word is unloaded from the scratch-pad output data register. Other applications may make it desirable to have more elaborate accessing modes which provide stack, queue, reverse stack, and reverse queue action. As one would expect, there is a significant trade-off between logical simplicity and scratch-pad capability. Also as elaborate structures are considered one soon realizes that the required setup activities become significant so that frequent mode switching causes drastic reduction in throughput.

It was found that for the emulator described in chapter 4, a scratch-pad memory with autoincrement addressing was desirable. More powerful capabilities would not have been sufficiently utilized to warrant the extra cost. The scratch pad memory utilized in chapter 4 will be described.

The scratch memory unit (SMU) has the capability of automatically incrementing the address register. Because of this, the address register need only be loaded once when a series of accesses will be performed at sequential locations. The effective speed-up produced by this arrangement approaches a factor of two as the string of accesses becomes long. This is due to the fact that only one bus cycle per memory reference is required. It is assumed that the memory speed of the SMU is high enough to allow access time to overlap bus cycles.

Figure 2.17 illustrates the auto-increment SMU. The memory controller causes the address register to be incremented at the proper times, and supplies local interlock signals to the source and destination controllers

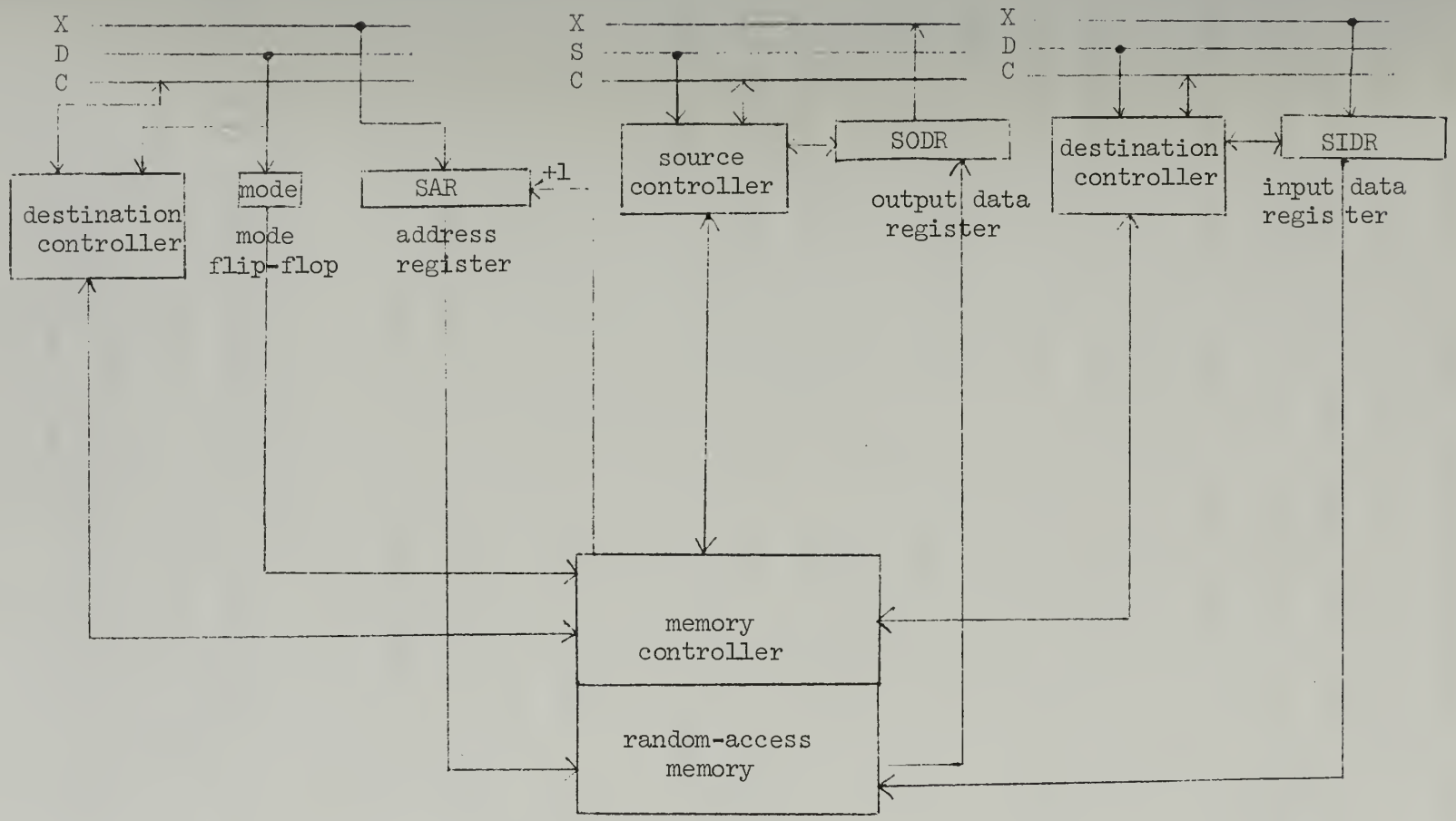
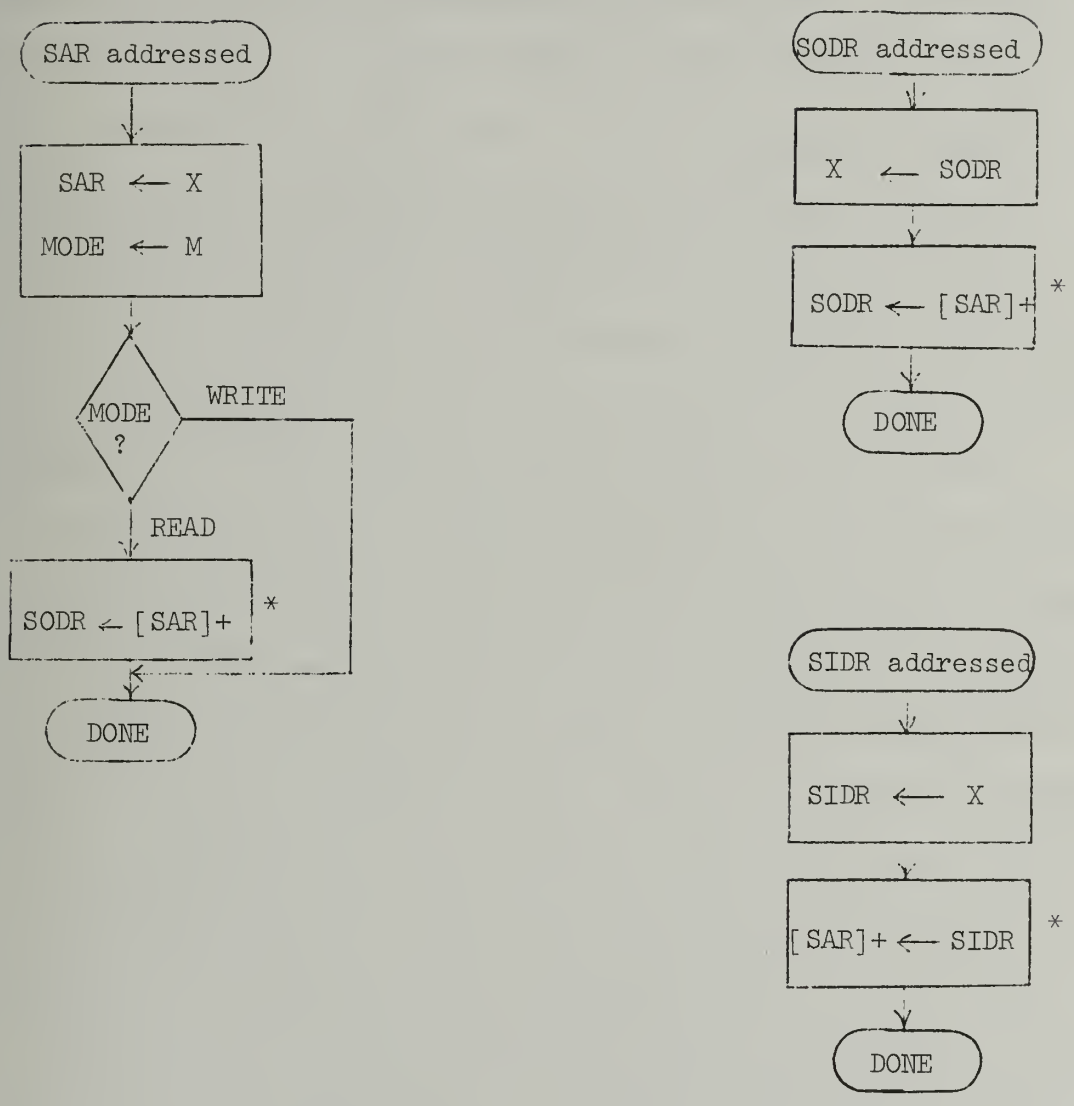


Figure 2.17 Auto-increment Scratch Memory Unit

in addition to controlling the memory. Operation of this SMU is as follows. Any memory operation is initialized by first loading the address register (SAR). When SAR is loaded the mode, read or write, is specified by one of the destination bits. If the mode is read, then a memory cycle at the address loaded into SAR is initiated, with the data appearing in the output data register (SODR). A subsequent bus cycle using the SODR as the source retrieves the desired data from the SMU. When this is done, the SAR is incremented and another read operation is initiated. This allows a series of bus cycles specifying SODR as the source to unload sequential words from the memory. If the mode specified when SAR is loaded is write, the memory cycle is not initiated until the input data register (SIDR) is loaded (from the bus). When SIDR is loaded a memory write cycle is initiated using the new contents of the SIDR as memory input. The SAR is incremented by this operation. Thus a series of bus cycles specifying the SIDR as the destination causes sequential memory locations to be loaded. Random access of the memory is accomplished by forgetting that the auto-increment is taking place, although it must be remembered that for memory write, the SAR is loaded first, then the SIDR. The mechanism described above also allows mixing of read and write operations at sequential memory locations. Figure 2.18 gives the algorithms implemented by the memory controller.

2.4.6 Main Memory

Implementation of main memory is very straight-forward. Data available and memory available signals are generated by the memory controller and are used by the bus interface controllers (address, data in, and data out), to form the local interlock conditions. Interleaved memory modules are an



*The "+" indicates increment SAR after the memory operation.

Figure 2.18 Scratch Memory Controller Algorithms

obvious extension. Main memory design for VK-1 is typical of that for other systems, and will not be further discussed.

In chapter 4, a total storage capacity of 2^{24} words is required. This necessitates adding an address extension register. Thus the loading of the address register requires two bus cycles.

2.4.7 External Interface

External interface and communications (I/O) is one area which has not been fully investigated by the research reported here. The VK-1 bus structure allows convenient transfer of data between bus devices, and therefore allows fairly powerful types of I/O to be implemented in microcode. External interface will be handled by a bus device which controls interlocked transfers to the "outside world." This controller could communicate either in serial, byte parallel, or word parallel fashion. Autonomous block transfer capability could be implemented by allowing the interface module to become bus master and assert source and destination addresses and execute a bus cycle. Such a capability would also require word count and extended status information to be maintained by the interface module. Hardwired microprogram interrupt capability is also a possibility. This could be done by having the interface module obtain the bus, store the current microinstruction counter and then execute a branch using its assigned interrupt vector as the source. (Macro-level interrupt is done in microcode by polling external interface module status registers at the appropriate times.) The design questions with respect to I/O are application dependent because they can, for the most part, be implemented either in hardware or microcode depending on the requirements. For this reason I/O will not be further discussed.

2.5 Multiple Bus Controllers

As mentioned previously, the major objective of the design of VK-1 is to achieve a highly flexible structure into which a wide variety of special purpose functional hardware units may be plugged. The devices described in section 2.4 are only a small number of the possible units which could be included in a system. It is natural to next consider the problem of connecting processors together. This concept evolves naturally as one considers more and more sophisticated bus devices. Recall that the discussion of I/O mentioned the possibility of block transfer capability by allowing the I/O module to act as bus master. The problem of allowing a bus device to issue bus commands is very similar to the problem of multiprocessor connection. In order for processors to be connected together in a useful way, provisions must be made for the communication of control information and data. Since both of these are transferred over the busses of a VK-1 processor, access to the busses of one processor by another processor gives that processor potentially any degree of control it wants to assert. Therefore a very natural way of connecting processors exists, namely by making one or more busses of one processor continuous with a bus or busses of another processor. As an example, if bus A of processor I is connected to bus B of processor II, and the branch control unit of processor I is on bus A, then processor II has the power to alter the microinstruction counter of processor I, and therefore can completely monitor and control its activities. There is no need to restrict the number of processors which can control one bus to two. That is in the above example, bus C of processor III could also be a continuation of bus A, B. Since processors may posses more than one bus, and since more than one BCU-type device could be

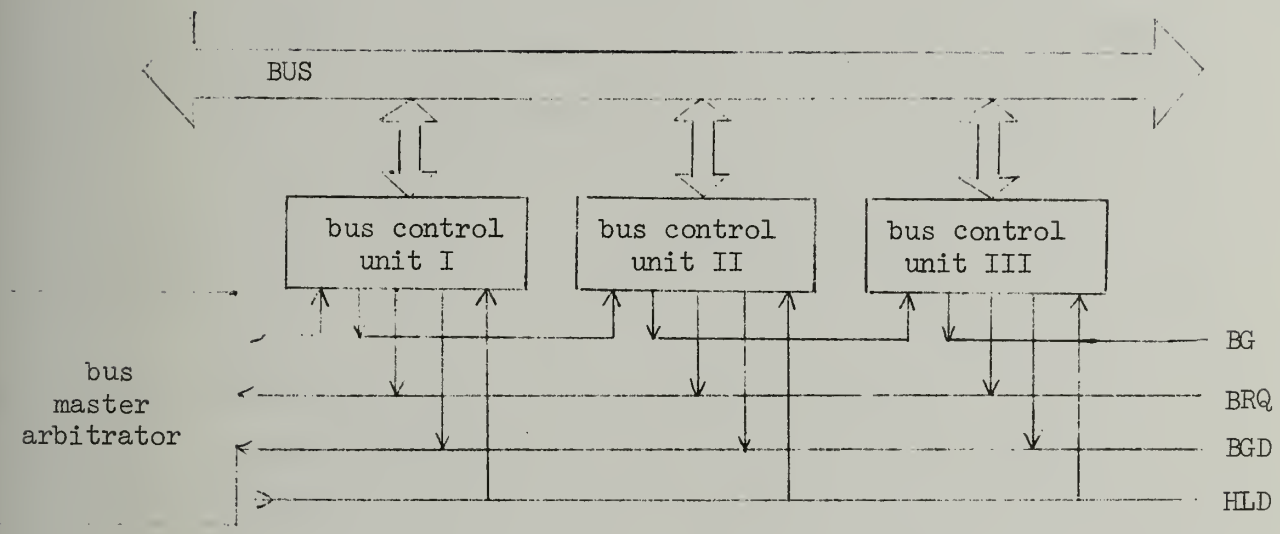
associated with a processor, the processor network configurations which are possible seem almost limitless. Further implications of such configurations are beyond the scope of this report.

One aspect of this idea which will now be discussed is the problem of arbitrating bus mastership when multiple bus masters are allowed. Figure 2.19 shows a bus master arbitrator and the additional control lines which must be added to the bus structure already presented. Figure 2.20 gives the flow diagrams for the bus controllers and the bus master arbitrator. The following activities take place. A device desiring bus control asserts BRQ. A device asserting BRQ gains control by capturing BG, a daisy-chained signal. (By adding more BG lines an arbitrary priority tree may be implemented.) A bus device which has gained control may override the other requesting devices by asserting BGD which inhibits the arbitrator from issuing BG. A device which has gained control maintains control until HLD clears (this can occur only when BGD is not being asserted). When HLD clears, control is lost and a new request must be issued to recapture control of the bus.

The arbitrator honors a BRQ by issuing BG and setting HLD anytime a BRQ occurs and BGD is false. The issuing of BG and the setting of HLD is synchronized with the bus cycle. Each bus device must maintain a status flag which tells it if it is in control of the bus.

2.6 Implementation Considerations

Detailed design and implementation has not been carried out in this research. An attempt has been made, however, to investigate the feasibility of the proposed hardware, and some observations have been made.



BRQ bus request
BG bus granted
BGD bus grant disable
HLD hold

Figure 2.19 Multiple Bus Control Unit Configuration

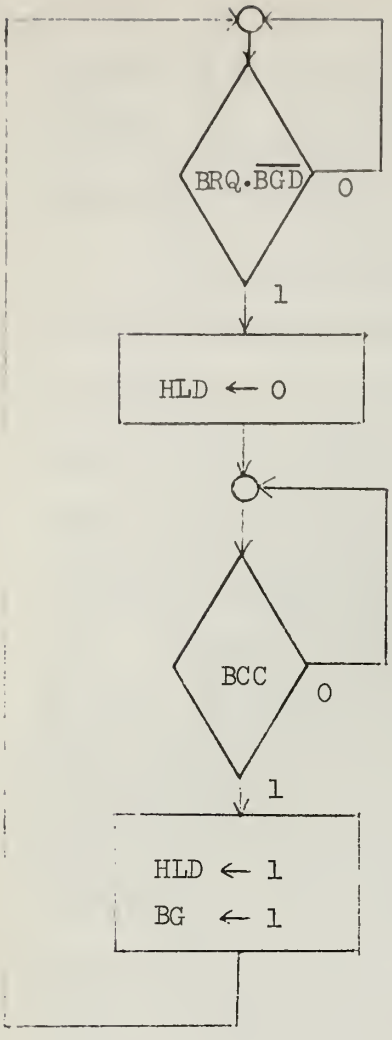
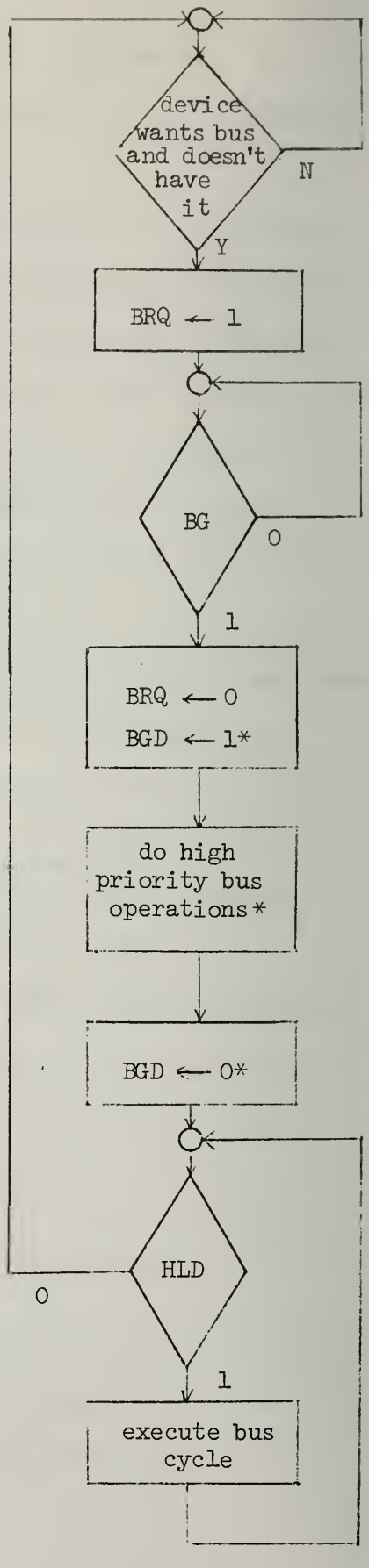


Figure 2.20a Bus Arbitration Controller



*option, depends on situation

Figure 2.20b Bus Device Interface Controller

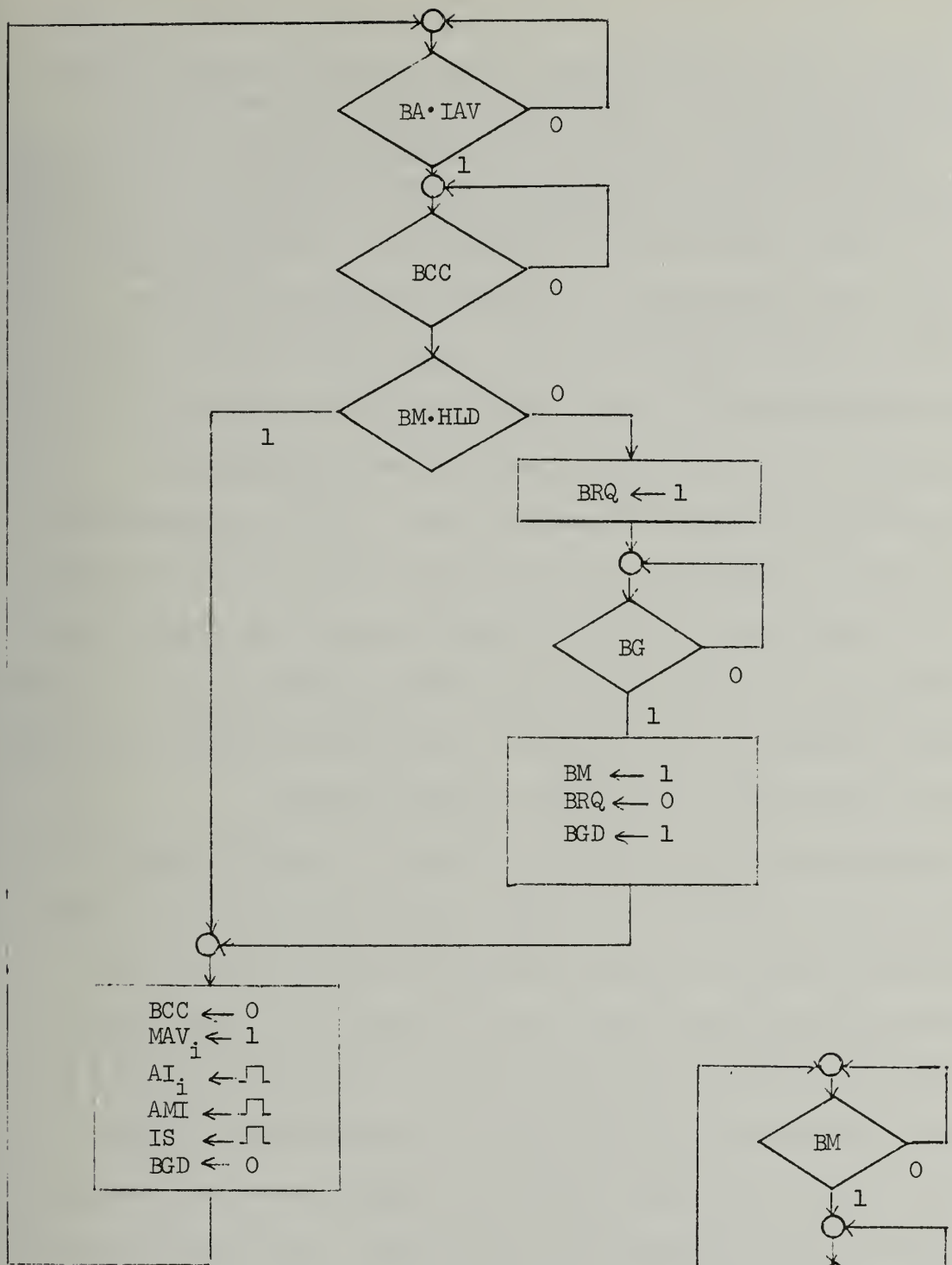


Figure 2.20c Instruction Strobe Controller

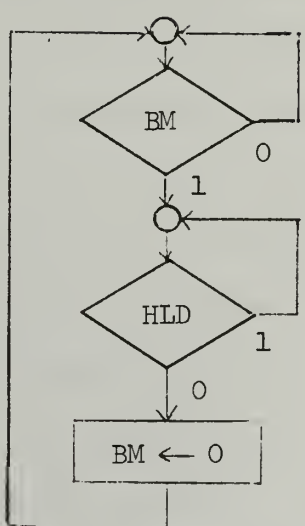


Figure 2.20d Bus Master Flag Controller

Ultimate speed and reliability of the system are heavily dependent on the bus implementation. In the author's opinion, a high speed bus of the sort required in VK-1, should provide controlled impedence paths for all signals, and these signals should be differential. An emitter-coupled logic (ECL) family which features very high input impedence has been announced by Fairchild Semiconductor [2]. Such a family should provide a very clean implementation of a fast, reliable bus. If, however, the speed/economics tradeoff dictates a cheaper slower implementation, transistor-transistor logic (TTL) could be used. In a TTL implementation one should note the difference between the S and D lines, and the transfer lines. The S and D lines are driven by single sources (when there is only one bus master per bus), whereas the X lines have multiple sources. Therefore, the tri-state approach [8] is a natural for the X lines, whereas conventional TTL could be used to drive and receive the S and D lines.

Special consideration must also be given to the multiplexor in the instruction stream generator. This could become a bottleneck since every microinstruction executed must pass through this unit. This unit will require very high performance logic such as schottky TTL. Since a small number of busses are being driven from the multiplexor, it should only be necessary to multiplex the control store lines and then drive the bus controllers in parallel. High speed circuits should be used to decode the bus address and clock the addressed bus controller. Also the circuit which detects immediate data instructions must be very fast since this must be checked by the multiplexor on every instruction issued.

Implementation of the bus devices described seems relatively standard. The bus interface circuit will be repeated many times in a system of reasonable size, and is therefore a candidate for custom MSI implementation if a production level design were considered.

Cost estimation has not been done since this is a very lengthy and tedious task, and can be misleading if done wrong. Obviously the decentralization of control and microinstruction decoding does not make for a cheap system.

3. SYSTEM SIMULATION

The planning stages of practically all computer design efforts include extensive simulations. Such simulation was carried out for VK-1. This section describes this effort.

3.1 Purpose of Simulation

Simulation of VK-1 was carried out at two levels. First, in order to verify that the proposed structure warrants the study reported in this paper, simulation was used to obtain data on the theoretical performance of the system. The second level of simulation was carried out to verify the logical specifications of the machine, and to provide a system model which could be used in further research to study VK-1 during execution of microprograms.

3.2 Simulator Implementation

Both levels of simulation were implemented in GPSS, a general-purpose discrete-time simulation system. GPSS was found to be a very powerful language for the type of simulation required. A detailed description of GPSS is found in reference 4.

3.3 Theoretical Performance Simulation Results

The term "theoretical," which is used for lack of a more precise term, refers to study of the intrinsic behavior of the system. This behavior is divorced from things such as precise module type and

configuration, and is not concerned with the microinstructions being executed. A very simple system model is used. This model incorporates as parameters the number of busses, bus cycle time, and control store band width. Since a fixed bus cycle time is used, microinstructions which activate modules which may be busy for more than a bus cycle are not considered. Therefore the model is not useful for estimating performance of real systems. The value of this model is that it determines the performance dependency on the above parameters, and estimates the gain in performance which is obtainable. This model is shown in figure 3.1.

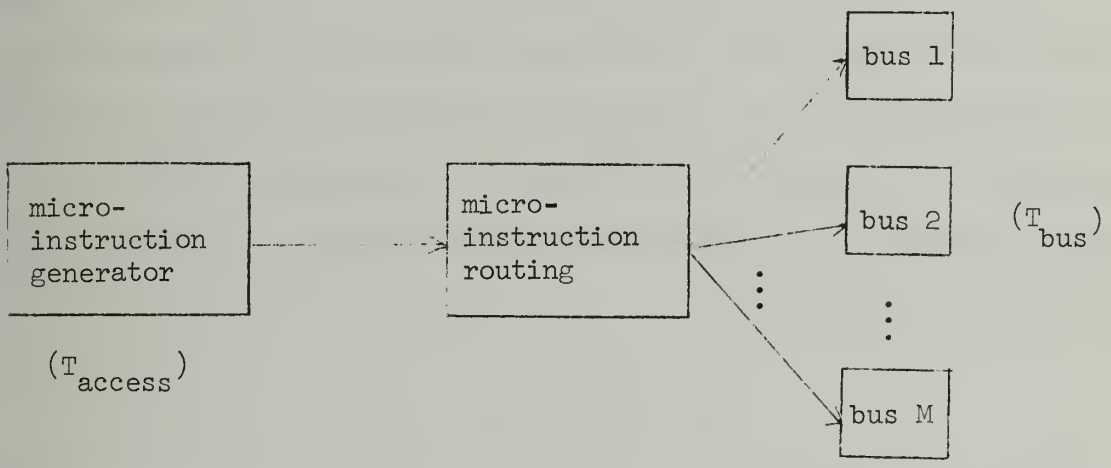


Figure 3.1 Model for Theoretical Performance Simulation

From the model it is obvious that if the microinstruction generator has sufficient band width, $\frac{1}{T_{access}} \geq M \frac{1}{T_{bus}}$, then optimally written microcode could be executed with a theoretical speed-up of

$$S_T = \frac{T\ (1\ bus)}{T\ (m\ busses)}$$

$$= \frac{T\ (1\ bus)}{T\ (1\ bus)/M} = M$$

This speed-up figure is not very valuable since it is nearly impossible to write microcode which uses the busses in an optimally scheduled way, however it is valid as an upper bound. A more reasonable model for the micro-instruction sequence is that which randomly schedules bus use. This model should provide reasonable performance gain estimates when bus address assignment optimization is not possible.

The GPSS program which creates and executes the above model with random bus addresses is shown in appendix A. Parameters for this program were swept to obtain the data illustrated in figures 3.2 and 3.3.

Figure 3.2 was obtained using Taccess = 0. This was done so that the data would reflect only scheduling collisions and not control-store band width limitations. This data illustrates that speedup increases at a decreasing rate as busses (and thus cost) are increased. From this diagram ~ 4 appears to be a reasonable number of busses, and with this number, the theoretical speedup is:

$$S_T = \frac{100}{45} = 2.22$$

Note that by doubling the number of busses from 4 to 8, speedup of only 3.31 is obtained.

Figure 3.3 shows the effects of control-store band width limited operation. This graph gives useful information for control-store design when the number of busses is known.

3.4 VK-1 Logical Simulator

A brief discussion of the operation of this program is impossible without assuming that the reader is familiar with GPSS, and therefore

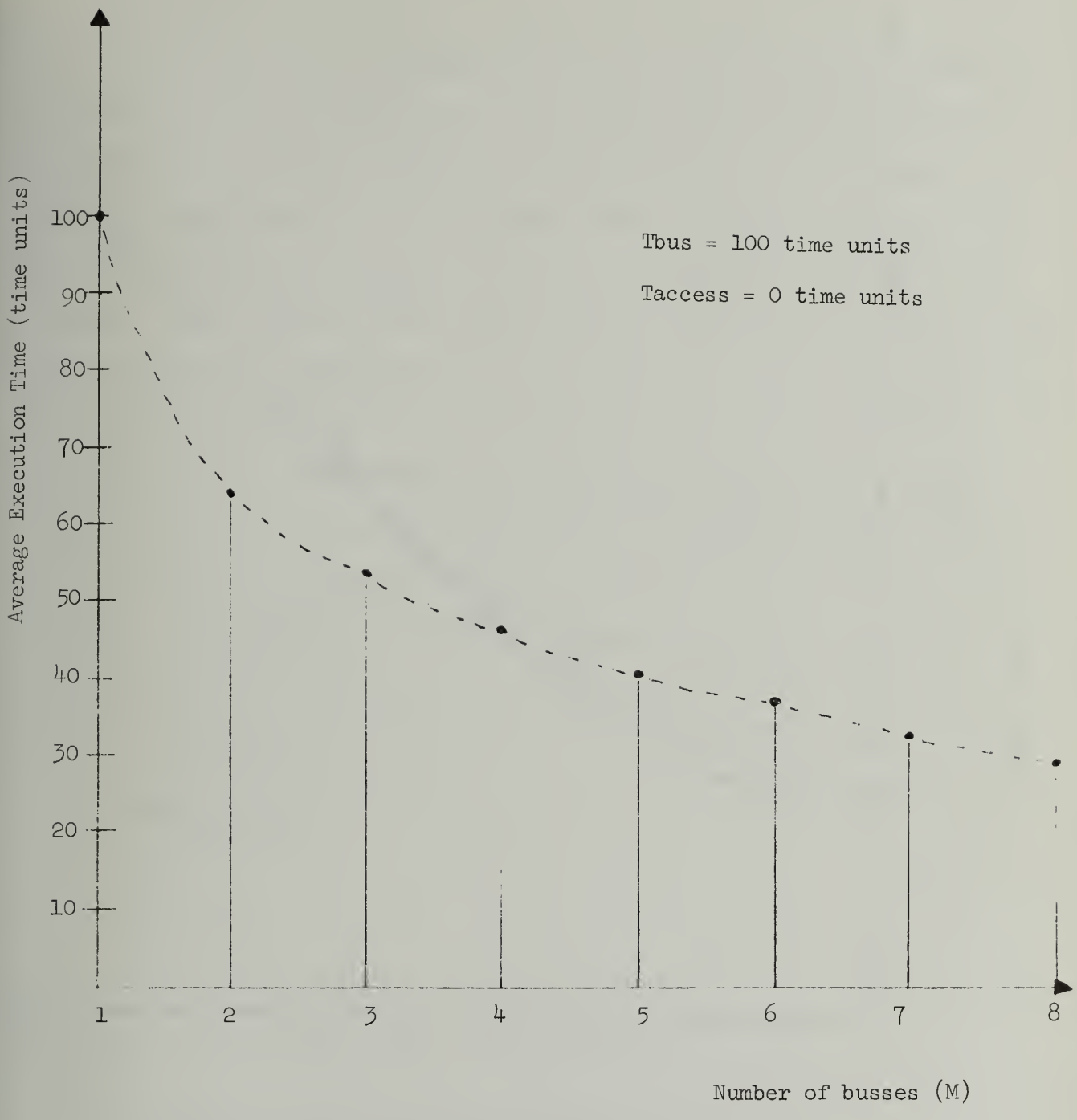


Figure 3.2 Execution Time vs. Number of Busses

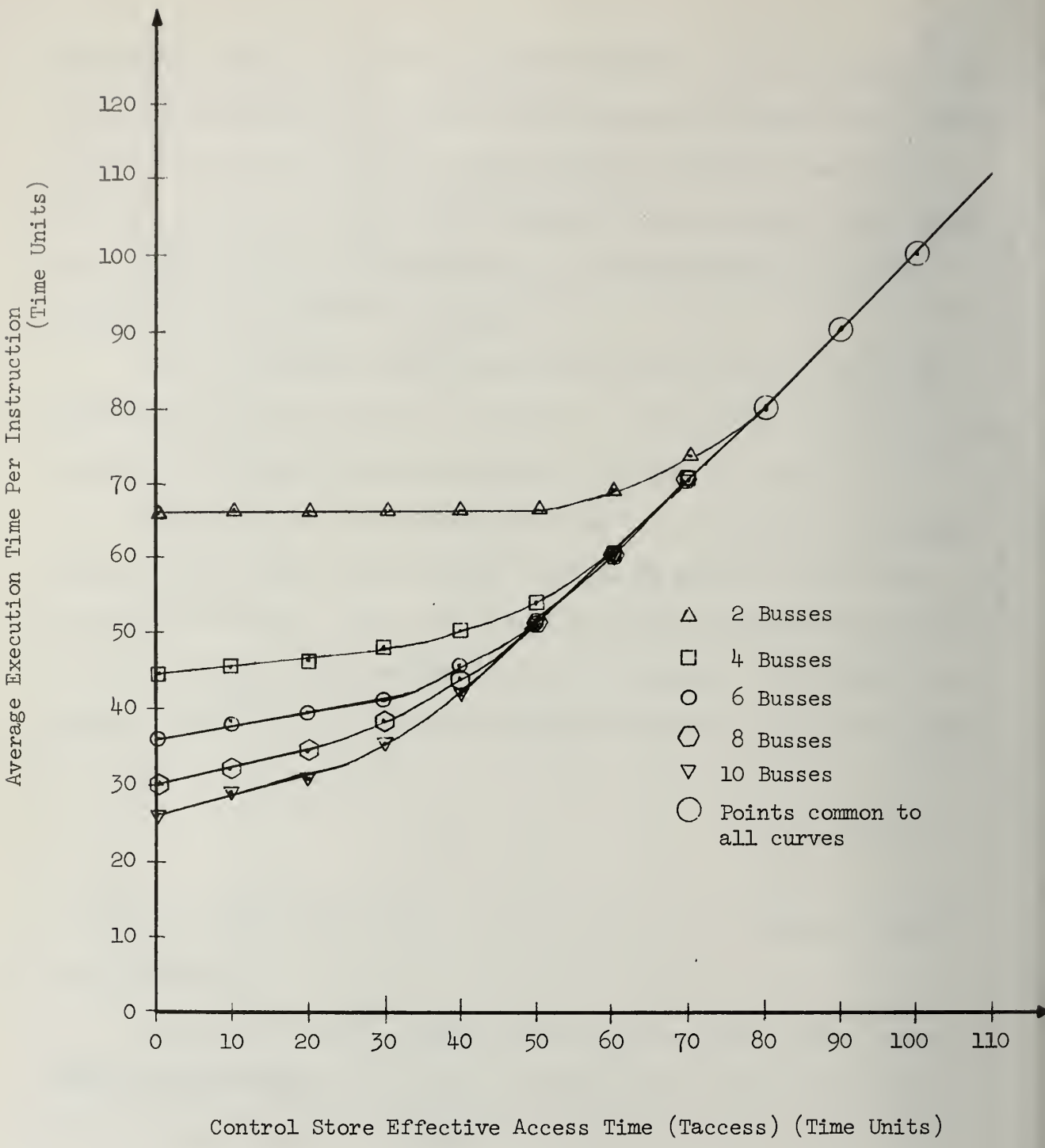


Figure 3.3 Execution Time Vs. Taccess

will not be attempted. The program has the capability for inputing micro-programs and loading core memory. Every control signal and logical activity is simulated, and timing parameters may be easily varied to reflect various implementation technologies. The program makes available an abundance of useful statistics such as utilization of control-store, busses, and bus devices. Queue statistics are also generated for busses and bus devices. These statistical outputs make the logical simulator a very powerful design tool allowing the designer to detect bottlenecks, "tune" the system, and optimize bus device assignments. In addition the logical simulator may be used to debug microcode and even macrocode. The simulator was very useful in the design work reported in chapter 2, and will be very useful in continued VK-1 research. A listing of the logical simulator is given in appendix B.

4. MICROPROGRAMMING VK-1

The success or failure of the proposed computer structure hinges on the obtainable rate of throughput for practical problems. The microprograms which control the information processing must be organized such that they take advantage of possible concurrence in bus operations. Optimization of a firmware system therefore requires maximizing bus-cycle overlap. The development and optimization of firmware will be discussed, and a detailed example of a large microprogram will be presented.

4.1 Firmware Development

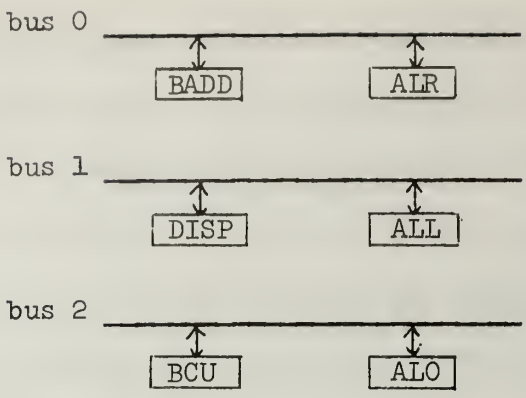
Development of VK-1 firmware is a two phase operation. Phase 1 is the selection of hardware bus devices which will be used, and the generation of instruction streams which accomplish the desired data and control manipulations utilizing these hardware devices. This phase may, in general, take place without regard for the physical bus configuration of the selected bus devices. Phase 1 is then, in this sense, configuration independent, and can be carried out with relatively little knowledge of the actual VK-1 device configuration. Also this hardware independence means that an instruction stream developed for a certain set of hardware modules may be used as phase 2 input for many system configurations. For example phase 2 could be applied three times to the same phase 1 output to produce firmware for one, two, and four bus machines. This concept will become clear when phase 2 is discussed.

As an example of the phase 1 activity suppose one wished to create a procedure which will select and enter one of many subprocedures by computing a control storage address and then jumping to it. A set of the following hardware devices might be used: two bus registers (symbolically referenced as BADD and DISP), an ALU and a BCU (of the types previously discussed). Then the following microinstruction stream would accomplish this:

SOURCE	DESTINATION	FUNCTION	COMMENT
BADD	ALR		ADD THE BASE ADDRESS
DISP	ALL	ADD2	TO THE DISPLACEMENT, AND
ALO	BCU	UC	JUMP TO THIS EFF. ADD

Phase 2 is the assignment of devices to busses, and the specification of a bus address for each instruction. Phase 2 is the critical stage of the firmware development because it is this assignment procedure which determines the amount of bus-cycle overlap. The problem surrounding phase 2 is that of assigning devices and bus addresses so that sequential instructions utilize different busses allowing overlap of the instruction set up. To clarify this notion, consider the simple instruction stream considered above in the discussion of phase 1. Figure 4.1 illustrates a hardware configuration which could perform the above function.

Obviously the first two instructions can be overlapped with this configuration as illustrated in Figure 4.2.



BUS ADD	SOURCE	DEST	FUNCTION
0	BADD	ALR	
1	DISP	ALL	ADD2
2	ALO	BCU	UC

Figure 4.1 Bus Assignment for Example

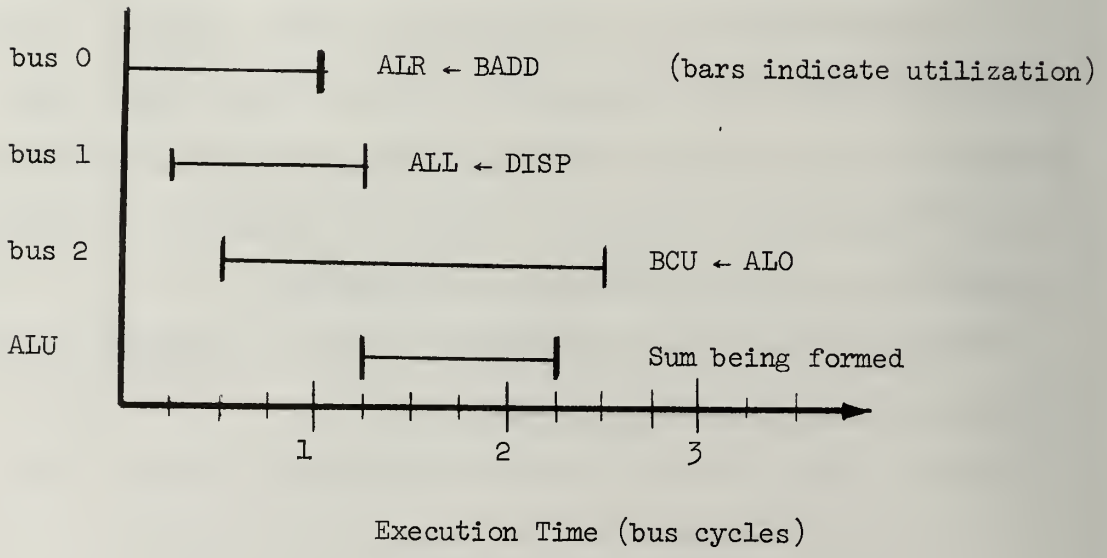


Figure 4.2 Overlap of Bus Cycles for Example

Unfortunately the problem of determining the optimal bus assignment for a given instruction stream of practical size becomes extremely complex for a general instruction mix. The maximum obtainable speed up is a factor equal to the number of busses. This theoretical maximum is virtually impossible to obtain unless the instruction stream exhibits highly regular patterns of sequences. An exact procedure for determining the optimal configuration would probably require exhaustive simulation of all reasonable possibilities. String analysis of the source-destination pairs comprising the instruction stream appears to be much too complex to be considered as an algorithmic approach to assignment optimization.

4.2 Symbolic Microprogramming Language

The language used to describe microinstructions is very simple and requires little explanation. This language bears considerable resemblence to the symbolic assembler language of most two address computers. The following format is used for all microinstructions:

LABEL BUS ADDRESS SOURCE DEVICE DESTINATION DEVICE FUNCTION COMMENT

The LABEL and COMMENT fields are optional, and the FUNCTION field may or may not be required depending on the particular instruction. The bus address, source device, and destination device fields are required for every instruction. Immediate data, that is a data word which is stored in the control store, is identified by the presence of enclosing single quotation marks. These are the only format requirements which were used in preparing microcode. Device and function mnemonics depend on the particular system and will be presented with the microprogramming example of the next sections.

4.3 Emulation of the IBM System/360 on VK-1

In order to further evaluate the design concepts of VK-1 from a microprogramming viewpoint, and to determine if the device types considered are adequate it was felt that a considerable microprogramming effort should be made. Emulation, the major application area of microprogramming, was chosen. In order to have a bench mark for comparison of the hardware and firmware, it was felt that firmware for emulating a familiar system should be developed. As a result of these considerations, a partial emulator for the IBM system/360 line of general purpose computers was chosen.

Emulation may be loosely defined as the hardware/software interpretation of the instruction set of one machine, termed the target machine, by another, referred to as the host machine. In a microprogrammable host machine, this interpretation is accomplished by the firmware, which is referred to as the emulator. Design of an emulator involves the mapping of the target machine hardware into the host machine hardware. This mapping is required at the following four levels[3]:

(a) main storage

(b) I/O

(c) registers, accumulators, flip-flops, etc.

(d) all other addressable resources of the target machine

Obviously the more similarity that exists between target and host, the less complex is this mapping, and in general, the more efficient the emulator. In selecting the 360 as a target machine it was realized that close similarity does not exist. It was felt that this was desirable because this would allow a fair evaluation of VK-1 as a general-purpose

user-microprogrammed machine. Also selection of such a target would illustrate the use and configuration of various bus devices for achieving an efficient host for a specified target, which is a major design goal.

4.3.1 VK-1 Hardware used in 360 Emulation

The adaptability of the system hardware to the specific application was not exploited to the fullest extent possible for the emulator which was developed. The bus devices utilized are the following:

4-port ALU
4-port shifter
core memory module 2^{24} (max) x 16 bit
64 word scratch pad memory with auto increment
branch control unit
11 general purpose bus registers

Additional hardware would have been desirable if the full system/360 had been emulated. For example a storage key memory module would have been essential for rapid operation. The 360 I/O would have also required more sophisticated hardware for efficient operation. It is interesting to note that a significant portion of the internal instruction set has been emulated with a very modest hardware configuration.

The ALU is the busiest device in the firmware developed, and so it was found to be essential that the ALU be accessible to all busses. Without this feature it becomes very difficult to take advantage of possible bus cycle overlap.

4.3.2 Emulator Organization

The mapping of the target machine into the host machine is illustrated in figure 4.3 and simply involves assigning the addressable storage units of the 360 to storage within VK-1. Frequently processed

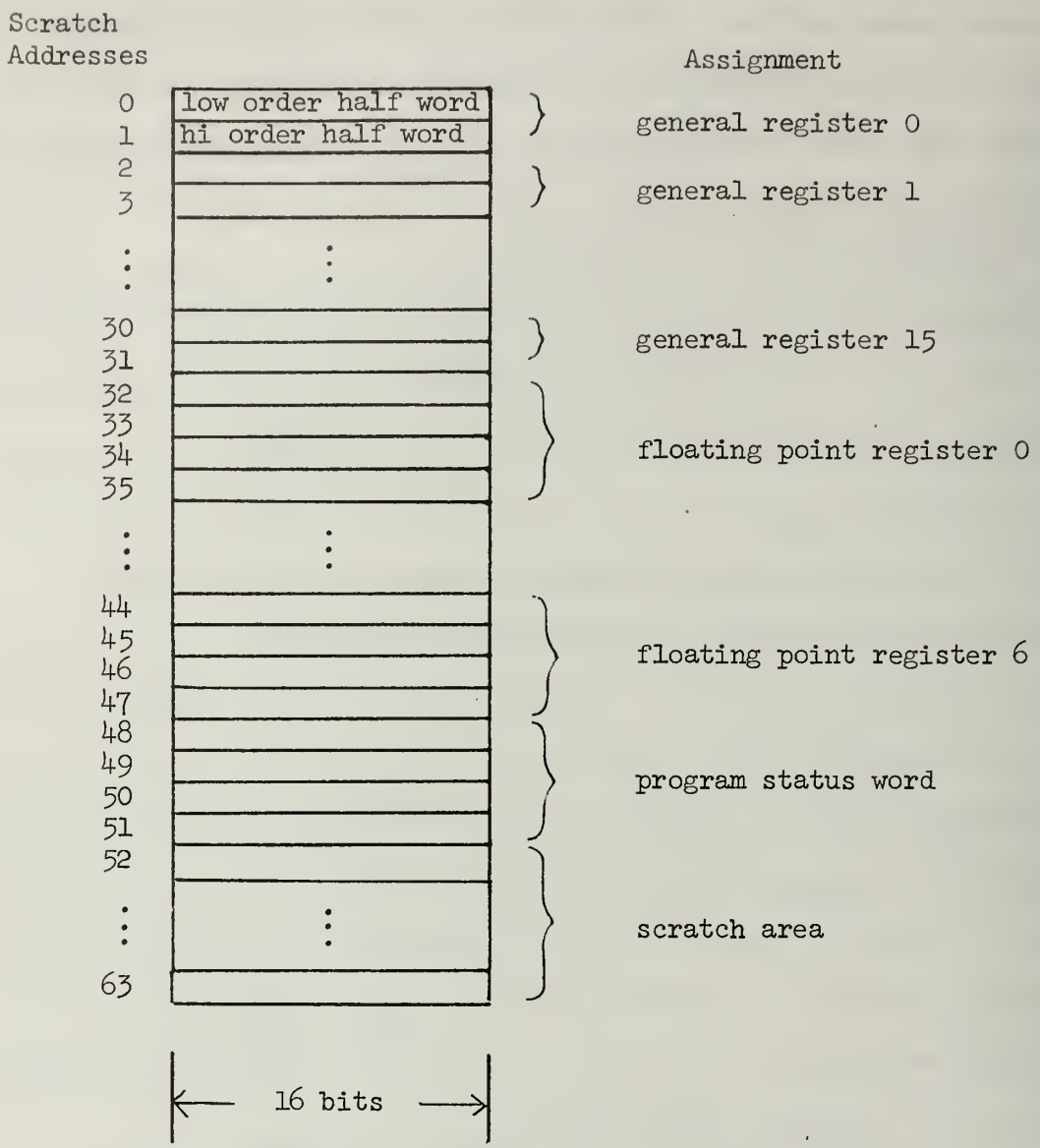


Figure 4.3 Scratch Memory Assignment

information is stored in bus registers, which are accessible in one bus cycle. These bus registers and their symbolic names and functions are represented in table 4.1. The mnemonics are used only for convenience in writing and understanding the emulator, and the format defined in section 4.2 is adhered to throughout the microprogramming discussion.

Register Mnemonic	Bus Add.	Function
ICLO	3	low order half-word of instruction counter
ICHI	2	high order byte of instruction counter and program mask
IRO	1	low order half-word of the current instruction
IR1	1	high order half-word of the current instruction
EALO	3	low order half-word of the effective address to be used next
EAHI	2	high order byte of the effective address to be used next
OP2LO	2	low order half-word of the second operand
OP2HI	2	high order half-word of the second operand
LKRG	0,3	linkage register for subroutine linkage
CCR	0	condition code register (holds internal condition codes)
T1	3	temporary storage register

Table 4.1 Bus Register Utilization

The overall program flow of the emulator is shown in figure 4.4. Even though only a subset of the system/360 was emulated, the approach taken was to organize the emulator so that the complete 360 could be emulated by addition of microroutines without change to what has already been developed. Since all of the "hooks" for a full emulator are present, the execution times for the instructions implemented would not be effected by the extension of the emulator to implement the full 360. For example, decoding capability for all instruction types has been included, even though execution routines have been written only for the implemented subset.

In firmware design one is frequently confronted with a design tradeoff between execution speed and control store usage. Even though VK-1 has microaddress space for 64K words of control storage, a considerable attempt was made to conserve control storage, as this is still one of the more expensive system resources. Commensurate with this objective, subroutining and shared code are used frequently. Subroutine linkage is not automatic in VK-1, so that the calling routine must store a return address in a specified memory device to facilitate return from the subroutine. A bus register is dedicated for this purpose, and has symbolic name LKRG.

The most frequently used fields of the program status word (PSW) are stored both in scratch memory and bus registers. These fields are the instruction counter, the program mask, and the condition code. The condition code field of the PSW is two bits requiring the four possibilities to be encoded. Since the condition code is infrequently required in this encoded form, its bus register form is not encoded. This allows the program to store the internal condition codes of VK-1 rather than converting

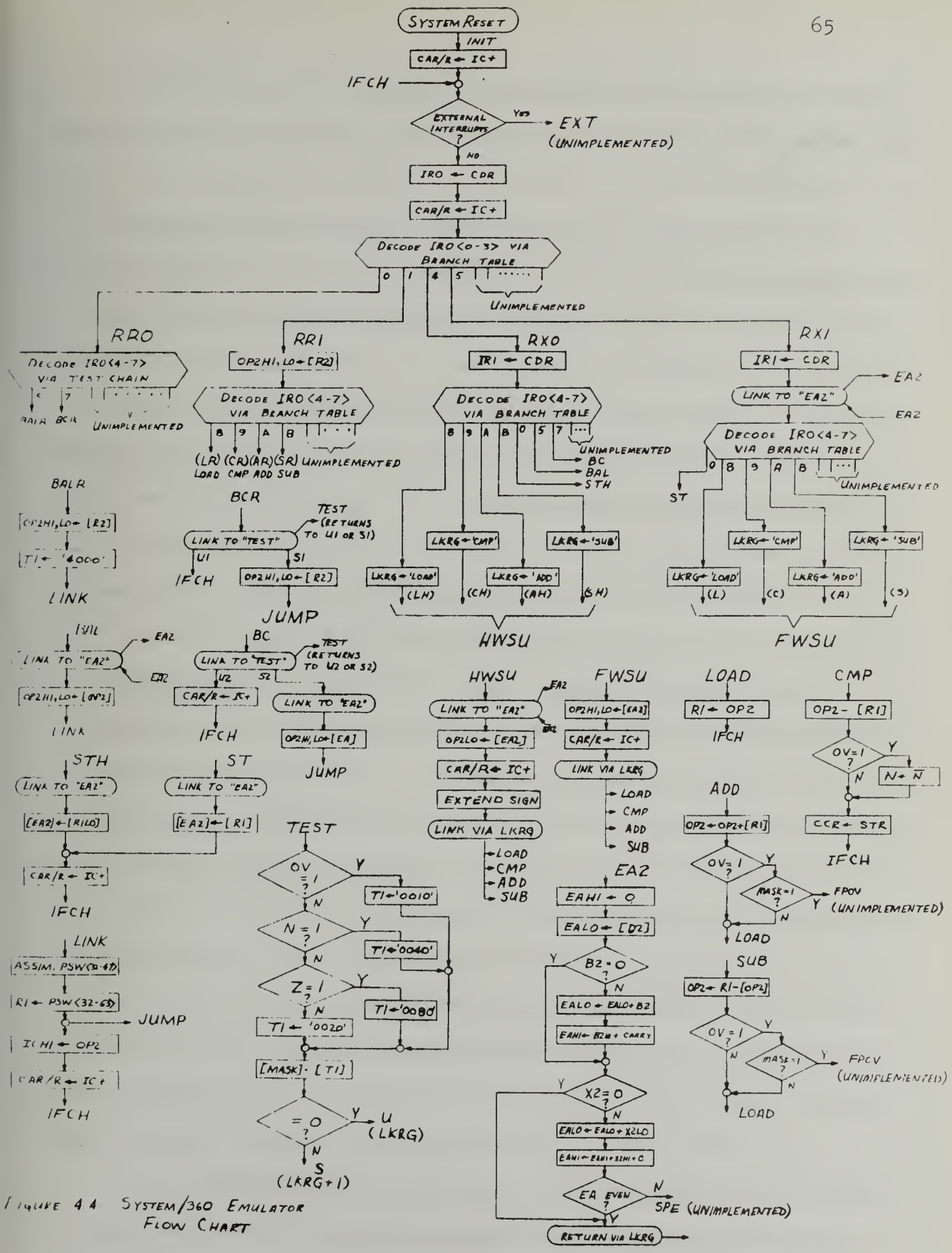


FIGURE 44 SYSTEM/360 EMULATOR FLOW CHART

to 360 condition codes for all instructions which alter the condition code. Then whenever the explicit 360 condition code is required it is assimilated from this internal form.

The main memory module is the slowest device in VK-1, thus it is important to maximize the utilization factor for main memory. This objective is met by initiating memory cycles at the earliest possible time. It is felt that the main memory utilization factor is a good measure of emulator efficiency, with main memory bandwidth limited operation as the ultimate goal. If this situation is obtained, the next step would be increase memory bandwidth by techniques such as increasing the word size accessed or interleaving several memory modules.

To further clarify the operation of the 360 emulator, the following descriptions of the major program segments are presented:

IFCH (instruction fetch). This section first checks for external interrupts pending, and if any exist, control is transferred to EXT for handling. (External interrupt handlers were not implemented.) The next half-word (16 bits) of the target program is loaded from the main memory data register (CDR) into IRO (instruction register 0). The memory cycle which accessed this word was initiated before IFCH was entered. The next half-word of the target program is accessed, and the instruction counter (ICHI, ICLO) is incremented. A branch table decoder causes control to transfer to one of 16 routines for second level decoding. This decoding is based on the low order bits (0-3) of the 360 instruction, and thus decodes both format and type specification. Four of the 16 possible branch points are invalid operation codes, and error routines for these choices are entered if they occur. Three different operation code error routines

are required, (OPER2, OPER4, OPER6) and they correspond to 2, 4, or 6 byte instruction formats. Second level decoders for the RR0, RR1, RX0, and RX1 instruction types were implemented.

RR0 (register-to-register format, type 0). This section does a serial decoding to one of seven execution routines. The decoder checks for opcodes with high frequency of occurrence first. Execution routines for BALR and BCR instructions have been implemented.

RR1 (register-to-register format, type 1). This section completes the setup required for execution of the individual instructions. This setup consists of accessing the 32 bit second operand which is held in scratch memory and storing it in bus registers OP2LO, OP2HI. The final decoding is then completed, transferring to individual execution routines. The LR, CR, AR, and SR instructions were implemented.

RX0 (register-and-indexed storage, type 0). This routine first obtains the second half word of the instruction (already accessed by IFCH) from main memory (CDR) and stores it in bus register IR1. A branch table is used to complete the instruction decoding. The RX0 branch table differs from others used in that two instructions correspond to each operation code. The first saves the address of the execution routine for that particular instruction (e.g., 'ADD' is saved for the AH instruction), and the second word transfers control to a shared half word format setup routine (HWSU). HWSU will then exit to the appropriate execution routine. Not all instructions will require the two instructions so that some unused words of control store will exist in the RX0 branch table.

RX1 (register-and-indexed storage, type 1). This routine obtains the second half word of the current instruction from main storage data register (CDR). A subroutine named EA2 is then executed to compute the

effective address of the second operand. The RX1 branch table is next entered to carry control to the final execution routines. This branch table like that of the RX0 routine saves the execution routine address corresponding to the present instruction in the linkage register LKRG, then jumps to shared routine FWSU which does the setup required for full word operations.

The remainder of the subprograms which comprise the emulator are execution routines used to complete the various instructions, or are subroutines used by execution routines. They will not be described in detail because the operation is well documented by the comments of the individual instructions in Appendix C. Much sharing of microcode is used in the execution phase of the emulator. For example LR, LH, and L all use the same execution routine, and similarly for add, subtract and compare. The symbolic microcode for the System/360 emulator is given in appendix C. Table 4.2 gives the device mnemonics used in the emulator code. The function mnemonics used are the same as those used in the descriptions of the various devices and are not repeated here. Bus register mnemonics were presented in table 4.1.

As a further aid to understanding the operation of the S/360 emulator, figure 4.5 depicts the interaction between and flow through the various program modules.

4.3.3 Bus Assignment for Emulator Microprogram

As discussed in section 4.1, microprograms development for VK-1 is a two phase process. Section 4.3.2 has discussed the phase one development, the generation of an instruction stream which executes the desired data transformations. The second phase, that of bus assignment is the present problem to be addressed. As was pointed out above, this phase is critical

Type *	Mnemonic	Description
D	ALR	right input to arithmetic/logic unit
D	ALL	left input to arithmetic/logic unit
S	ALO	output of arithmetic/logic unit
S	SHR	output or input of shifter
D	CARH	high order bits (8) of address for main memory
D	CARL	low order bits (16) of address for main memory
S,D	CDR	main memory data (buffer) register
D	SAR	address register for scratch pad memory
S	SORD	output data register for scratch pad memory
D	SIDR	input data register for scratch pad memory
D	BCU	branch control unit
S,D	STR	system status register

* S means the device is a source

D means the device is a destination

S,D means the device may be either a source or destination

Table 4.2 Device Mnemonics used in Emulator

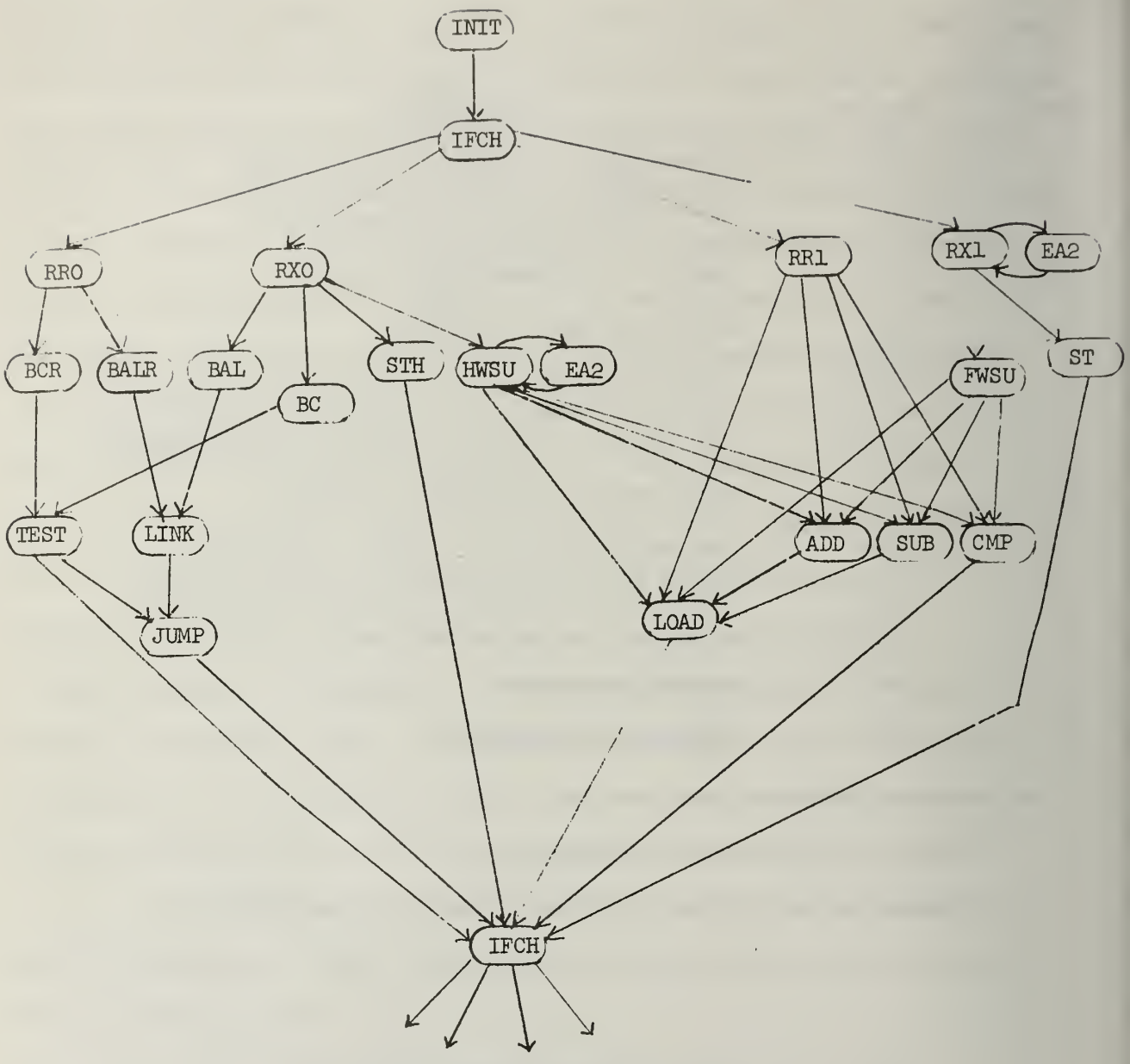


Figure 4.5 Flow of Control Through Emulator

to the development of firmware capable of utilizing to advantage the computer structure under investigation. Since the optimal assignment is program dependent, the first step in approaching the assignment problem is to analyze the instruction stream to be executed. An inspection of the emulator code immediately reveals the following: First there is not a significant number of repeated sequences of source-destination pairs, second, instructions utilizing the arithmetic and logic unit appear with very high relative frequency. Consequently to achieve a worthwhile amount of bus concurrency, a 4 port arithmetic and logic unit is essential. Since the type of firmware considered here possesses neither inherent parallelism nor repetitive sequences, further optimization of the hardware configuration is not attempted.

Table 4.3 contains one possible hardware configuration for execution of the System/360 emulator. This configuration was heuristically arrived at based on familiarity with the microcode. It attempts to evenly distribute the devices among the busses, and minimize interbus transfers. In this table, symbolic names for the bus registers are used to lend more understanding to this choice of device assignment. Note that in table 4.3 only those devices which are required for execution of the emulator routines are included. Input/output bus devices for example have not been considered in this emulation study.

4.3.4 Emulator Performance Analysis

The performance of the System/360 emulator may be evaluated analytically. This analysis yields instruction times for the VK-1 system, and also reveals some of the inherent problems of the structure.

Bus Address	0	1	2	3
	ALL	ALR	ALL	ALR
	ALO	ALO	ALO	ALO
	SHR	SHR	SHR	SHR
device mnemonic	STR*	SAR	CARH	CARL
			SDDR	UCOM
	BCU	CDR	SDR	ICLO*
			SIDR	
	UCOM	UCOM	UCOM	
	CCR*	IRO*	ICHI*	EALO*
	LKRG*	IR1*	EAHI*	T1*
	SODR		OP2HI*	LKRG*
	LKRG*		OP2LO*	

*registers

Table 4.3 360 Emulator Bus Assignment

The analysis procedure used is simply drawing a hardware scheduling diagram for the microcode, using the constraints explained in chapter 2. Table 4.4 gives the timing parameters used in the scheduling procedures.

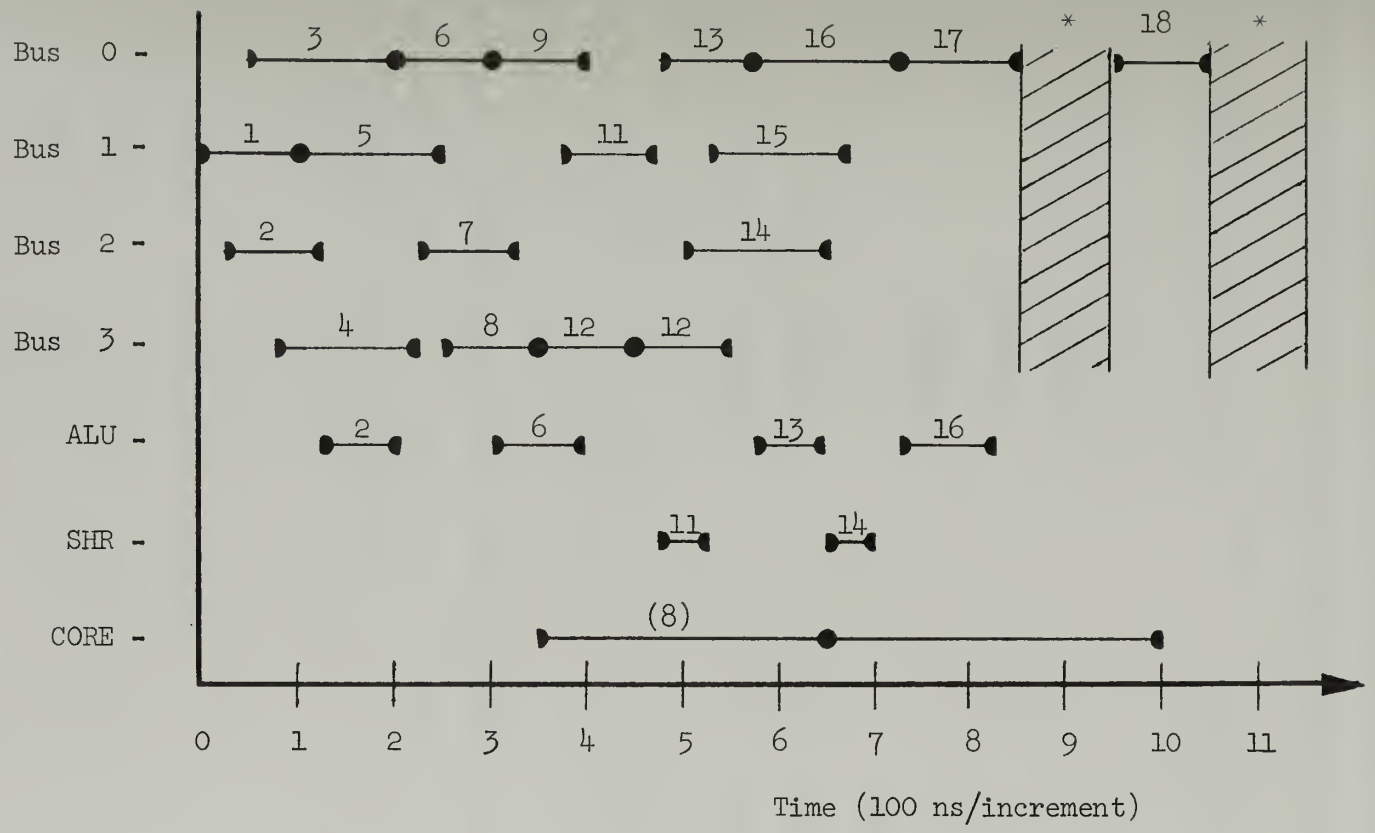
Activity	Time Required (ns.)
minimum bus cycle (no conflicts)	100
instruction multiplex time (min. time between instructions)	25
control store access time (per module)	75
arithmetic ALU operation	100
logical ALU operation	75
shifter operation (any distance)	50
core memory access time	300
core memory cycle time	650
scratch memory access time	50

Table 4.4 Timing Parameters Used in Analysis

Consider the program segment "IFCH." This code is shown in table 4.5. The scheduling diagram of figure 4.6 shows how execution of "IFCH" proceeds. It is assumed that a carry from the low order 16 bits to the high order 8 bits of the instruction counter does not occur when the instruction counter is incremented, and that no external interrupts are pending.

Statement No.	Bus	Source	Destination	Function
1	1	literal	ALR	
2	2	STR	ALL	logical
3	0	literal	BCU	unsuccessful
4	3	ICLO	UCOM	
5	1	CDR	IRO	
6	0	UCOM	ALL	arithmetic
7	2	ICHI	CARH	
8	3	ICLO	CARL	read
9	0	literal	BCU/1	unsuccessful (1 post inst)
10	3	ALO	ICLO	
11	1	IRO	SHR	shift
12	3	literal	ALR	
13	0	SHR	ALL	logical
14	2	ALO	SHR	shift
15	1	literal	ALR	
16	0	SHR	ALL	arithmetic
17	0	ALO	BCU	unconditional
18	0	literal	BCU	unconditional

Table 4.5 "IFCH" Microcode



Bars represent utilization.

*Shaded area represents control store latency due to successful branch.

Figure 4.6 Scheduling Diagram for "IFCH"

Table 4.6 shows the results of similar analysis on other program segments, and table 4.7 gives instruction times for bench-mark instructions. Also listed are the corresponding instruction times for the IBM 360 model 40 (a 16 bit machine)[6] and the IBM 360 model 50 (a 32 bit machine)[5].

Microcode Segment	Execution Time* (Micro seconds)
IFCH	1.150
RR1	0.925
RX1	0.825**
ADD	0.925
EA2	1.850
FWSU	1.650
LOAD	0.475

*Most frequent trace is assumed.

**Excluding time of subroutine "EA2".

Table 4.6 Execution Times of Emulator Segments

	Execution Time (micro seconds)		
Target Instruction	VK-1	360/50	360/40
LR	2.55	2.50	7.5
AR	3.475	3.25	7.5
L	5.9	4.00	11.88
A	6.825	4.00	11.88

Table 4.7 Comparison of Instruction Times for VK-1 and IBM 360/40, 50

These results show that reasonably good execution times are obtainable for general purpose microcode. The fact that the hardware structure of VK-1 was not planned with system/360 emulation in mind is significant. Most currently available microprogrammable processors perform well for a single type of target architecture, but degrade significantly when applied to other applications. Such performance degradation is avoided in VK-1 because the structure is general purpose, and because the system configuration is easily modified to accommodate the specific application. It should also be apparent to anyone who has written microcode that VK-1 is probably easier to code than any other microprocessor currently available.

The component speeds indicated in table 4.4, and the amount of hardware assumed in the emulator system might lead one to expect better performance. The price paid for a simple microcode structure utilizing a short and highly encoded format is inefficient utilization of the hardware. Since there are no directly controllable gating paths, a 100 ns. bus cycle is required for every system activity. The hardware utilization becomes worse when bus scheduling conflicts arise. This is to be expected when such a high degree of encoding is used.

5. CONCLUSIONS

The inherent structure of VK-1 does meet the major design goals of high performance, flexibility and ease of microprogramming. Hardware costs are high due to the use of redundant control and decoding circuitry. It is felt that this criticism of the VK-1 structure will become of decreasing importance as circuit costs continue to drop.

Once a microinstruction stream is programmed, the optimization of functional module-to-bus assignment is very difficult. For this reason it is expected that the random model simulation discussed in chapter 3 is fairly realistic for practical problems. This point is well illustrated by the results of chapter 4. Better performance, possibly approaching the upper bound of chapter 3, is more easily obtained when the microcode contains regular patterns of instructions allowing bus assignment optimization.

The relatively high "level" of the VK-1 microinstruction could probably be better exploited in applications such as the direct execution of high level languages. The use of multiple processor configurations also allows some interesting possibilities. The use of a VK-1 type structure as a machine language emulation engine is of dubious value, especially when cost is considered.

LIST OF REFERENCES

[1] Author unknown, "Microprogramming: The Way of the Past?", Computer, March/April, 1972.

[2] Fairchild Semiconductor, "9500 Series High Speed Logic," Fairchild Semiconductor, Inc., May, 1971.

[3] Husson, Samir S., Microprogramming Principles and Practices, Prentice-Hall, Inc., 1970.

[4] International Business Machines Corporation, "General Purpose Simulation System/360 User's Manual," Fifth Edition; January, 1970.

[5] International Business Machines Corporation, "IBM System/360 Model 50 Functional Characteristics," Second Edition, 1967.

[6] International Business Machines Corporation, "IBM Systems/360 Model 40 Functional Characteristics."

[7] International Business Machines Corporation, "IBM System/360 Principles of Operation," Ninth Edition; 1970.

[8] Morris, Mel, "National's Tri-State Logic," National Semiconductor, Inc., May, 1971.

[9] Ramamoorthy, C. V. and Tsuchiya, M., "A Study of User-Microprogrammable Computers," 1970 Spring Joint Computer Conf., pp. 165-181.

[10] Rosin, Robert F., "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, Vol. 1, No. 4, December, 1969.

[11] Wilkes, M. V., "The Best Way to Design an Automatic Calculating Machine," Manchester University Computer Inagural Conference, p. 16, 1951.

[12] Wilkes, M. V., "The Growth of Interest in Microprogramming--A Literature Survey, Comp. Surveys 1, (Sept. 1969), 139-145.

APPENDIX A

THEORETICAL PERFORMANCE SIMULATION PROGRAM

The following GPSS program implements the theoretical performance model discussed in section 3.3. Program operation is explained in the extensive statement comments given.

```
//JOBLIB DD DSNAME=SYS1.GPSSLIB,DISP=SHR
// EXEC GPSS
//GPSS.SYSIN DD *
      SIMULATE
*     THIS PROGRAM SIMULATES THE EXECUTION TIME OF THE VK-1 ORGANIZATION
* FOR RANDOMLY DISTRIBUTED BUS ADDRESSES.
*
*     FUNCTION 1 PROVIDES UNIFORMLY DISTRIBUTED BUS ADDRESSES.  A
* FACILITY AND A QUEUE ARE PROVIDED FOR EACH BUS TO GATHER THE
* DESIRED STATISTICS.  LOGIC SWITCH 'STEN' IS USED TO CONTROL THE
* CREATION OF MICROINSTRUCTIONS.
*THE NUMBER OF BUSSES AND THE ACCESS TIME OF THE CONTROL STORE MAY BE
* VARIED.
*
* ENTITIES UTILIZED BY MODEL:
* SERNO  EQU        1,H                    SERIALIZATION CONSTANT
* BUS1   EQU        1,F                    BUS1 FACILITY
* BUS2   EQU        2,F                    BUS2    "
* BUS3   EQU        3,F                    BUS3    "
* BUS4   EQU        4,F                    BUS4    "
* BUS5   EQU        5,F                    BUS5    "
* BUS6   EQU        6,F                    BUS6    "
* BUS7   EQU        7,F                    BUS7    "
* BUS8   EQU        8,F                    BUS8    "
* BUS9   EQU        9,F                    BUS9    "
* BUS10  EQU        10,F                   BUS10   "
* BUSQ1  EQU        1,Q                    BUS1 QUEUE
* BUSQ2  EQU        2,Q                    BUS2    "
* BUSQ3  EQU        3,Q                    BUS3    "
* BUSQ4  EQU        4,Q                    BUS4    "
* BUSQ5  EQU        5,Q                    BUS5    "
* BUSQ6  EQU        6,Q                    BUS6    "
* BUSQ7  EQU        7,Q                    BUS7    "
* BUSQ8  EQU        8,Q                    BUS8    "
* BUSQ9  EQU        9,Q                    BUS9    "
* BSQ10  EQU        10,Q                   BUS10   "
* STEN   EQU        1,L                    CREATION CONTROL SIGNAL
* CHNG1  INITIAL    LS1
* 1      FUNCTION   RN1,D4                 FN1 RETURNS BUS ADDS OF 1,2,3,4
*.25,1/.50,2/.75,3/1,4                     WITH EQUAL PROBABILITY
*
*********************SIMULATION MODEL*********************
*
* LOC    OPERATION  A,B,C,D,E,F,G           COMMENTS
*
* CHNG2  GENERATE   ,,,,,,2                 CREATE A MICROINSTRUCTION (T=0)
*        GATE LS    STEN                    CONTROL GATE OPEN? (IF NO--WAIT)
*        LOGIC R    STEN                    IF YES CLOSE GATE AND PROCCESS.
*        SAVEV      SERNO+,K1,H             UPDATE SERIAL NUMBER
*        ASSIGN     1,XH$SERNO              AND ASSIGN TO P1.
*        ASSIGN     2,FN1                   ASSIGN BUS ADDRESS TO PARM2
*        QUEUE      *2                      TAKE QUEUE STATS ON BUS USE
*        SEIZE      *2                      USE ADDRESSED BUS FACILITY
*        DEPART     *2                      EXIT FROM QUEUE
*        LOGIC S    STEN                    BUS OBTAINED--OPEN CONTROL GATE
*        ADVANCE    100                     DO CYCLE-- 100 TIME UNITS
*        RELEASE    *2                      GET OFF BUS
*        TERMINATE  1                       KILL, AND INCREMENT TERM COUNT
```

```
        START       1000                SIMULATE 1000 INSTRUCTIONS
        RMULT       1                                                  82
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    10,,,,,2            REPEAT SIMULATION WITH ACCESS
        START       1000                TIME = 10 TIME UNITS
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    20,,,,,2            T-ACCESS = 20
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    30,,,,,2            T-ACCESS = 30
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    40,,,,,2            T-ACCESS = 40
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    50,,,,,2            T-ACCESS = 50
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    60,,,,,2            T-ACCESS = 60
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    70,,,,,2            T-ACCESS = 70
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    80,,,,,2            T-ACCESS = 80
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    90,,,,,2            T-ACCESS = 90
        START       1000
        RMULT       1
        CLEAR
CHNG1   INITIAL     LS1
CHNG2   GENERATE    100,,,,,2           T-ACCESS = 100
        START       1000
        END
```

APPENDIX B

LOGICAL SYSTEM SIMULATION PROGRAM

The following GPSS program implements the logical simulator of section 3.4. It is a complex program utilizing many of the more advanced features of the GPSS simulation language. The extensive comments provided should explain the operation of the program to the experienced GPSS user.

```
//JOBLIB DD DSNAME=SYS1.GPSSLIB,DISP=SHR
// EXEC GPSS
//GPSS.SYSIN DD *
       REALLOCATE BVR,15
       SIMULATE
*    THIS PROGRAM SIMULATES THE OPERATION OF VK1.  THE PROGRAM CONSISTS
*OF MICROINSTRUCTION ASSEMBLER, IBUS CONTROL, BUS-MASTERSHIP ARBITRA-
*TION, COMPLETION QUEUEING, SOURCE AND DESTINATION DECODING, AND BUS
*DEVICE CONTROLLERS.
*
*
*MODEL ENTITIES DEFINITION: (SYMBOLIC NAMES MUST BE EXPLICITLY ASSOCI-
*ATED WITH DEVICE ENTITIES TO INSURE VALID OFFSET INDEXING.)
*
*
*HALFWORD MATRIX SAVEVALUES:
*
 RCM0  MATRIX     H,5,32              COLUMN = ADDRESS
 RCM1  MATRIX     H,5,32              ROW 1 = BUS ADD
 RCM2  MATRIX     H,5,32              ROW 2 = SOURCE ADD
 RCM3  MATRIX     H,5,32              ROW 3 = SOURCE FUNCTION
*                                     ROW 4 = DESTINATION ADD
*                                     ROW 5 = DESTINATION FUNCTION
*
*
*HALFWORD SAVEVALUES:
*
 MAR0  EQU        1,H                 MEMORY ADDRESS REGISTER 0
 MAR1  EQU        2,H                   "       "         "     1
 MAR2  EQU        3,H                   "       "         "     2
 MAR3  EQU        4,H                   "       "         "     3
 IMR0  EQU        5,H                 IMMEDIATE DATA REGISTER BUS 0
 IMR1  EQU        6,H                     "       "       "       "  1
 IMR2  EQU        7,H                     "       "       "       "  2
 IMR3  EQU        8,H                     "       "       "       "  3
 MEN   EQU        9,H                 MODULE ENABLE (MODULO 4 COUNTR)
 AMP   EQU        10,H                ACTIVE MODULE POINTER (MODULO 4)
 IMRP  EQU        11,H                POINTER TO IMMEDIATE DATA REG
*                                     TO BE LOADED WITH NEXT INST.
 MDIR  EQU        12,H                MEMORY DATA INPUT REGISTER
 XBUS0 EQU        13,H                XFER REGISTER--BUS 0
 XBUS1 EQU        14,H                XFER REGISTER--BUS 1
 XBUS2 EQU        15,H                XFER REGISTER--BUS 2
 XBUS3 EQU        16,H                XFER REGISTER--BUS 3
 MDOR  EQU        17,H                MEMORY DATA OUTPUT REGISTER
 MAR   EQU        18,H                MEMORY ADDRESS REGISTER
 ILLD0 EQU        19,H                ILLEGAL DESTINATION CODE FLAG
 ILLD1 EQU        20,H                ILLEGAL DESTINATION CODE FLAG
 ILLD2 EQU        21,H                ILLEGAL DESTINATION CODE FLAG
 ILLD3 EQU        22,H                ILLEGAL DESTINATION CODE FLAG
 ILLS0 EQU        23,H                ILLEGAL SOURCE CODE--BUS 0
 ILLS1 EQU        24,H                ILLEGAL SOURCE CODE--BUS 1
 ILLS2 EQU        25,H                ILLEGAL SOURCE CODE--BUS 2
 ILLS3 EQU        26,H                ILLEGAL SOURCE CODE--BUS 3
 REG00 EQU        27,H                REGISTER 0 -- BUS 0
 REG01 EQU        28,H                REGISTER 1 -- BUS 0
 REG02 EQU        29,H                REGISTER 2 -- BUS 0
 REG03 EQU        30,H                REGISTER 3 -- BUS 0
 REG10 EQU        31,H                REGISTER 0-- BUS 1
 REG11 EQU        32,H                REGISTER 1-- BUS 1
 REG12 EQU        33,H                REGISTER 2-- BUS 1
```

```
 REG13 EQU        34,H                REGISTER 3-- BUS 1
 REG20 EQU        35,H                REGISTER 0 -- BUS 2
 REG21 EQU        36,H                REGISTER 1 -- BUS 2
 REG22 EQU        37,H                REGISTER 2 -- BUS 2
 REG23 EQU        38,H                REGISTER 3 -- BUS 2
 REG30 EQU        39,H                REGISTER 0 -- BUS 3
 REG31 EQU        40,H                REGISTER 1 -- BUS 3
 REG32 EQU        41,H                REGISTER 2 -- BUS 3
 REG33 EQU        42,H                REGISTER 3 -- BUS 3
 UCOMR EQU        43,H                UNIVERSAL COMMUNICATIONS REG.
 ALLRG EQU        44,H                ALU--LEFT HAND REGISTER
 ALRRG EQU        45,H                ALU--RIGHT HAND REGISTER
 ALLFN EQU        46,H                ALU--LEFT FUNCTION REGISTER
 ALRFN EQU        47,H                ALU--RIGHT FUNCTION REGISTER
 ALORG EQU        48,H                ALU--OUTPUT REGISTERR
 SERNC EQU        49,H                SERIAL NUMBER FOR INSTRUCTIONS
*
*
*FACILITIES:
*
 ROM0  EQU        1,F                 RCM(FACILITY) 0
 RCM1  EQU        2,F                       "       1
 ROM2  EQU        3,F                       "       2
 RCM3  EQU        4,F                       "       3
 BUS0  EQU        5,F                 BUS(FACILITY) 0
 BUS1  EQU        6,F                       "       1
 BUS2  EQU        7,F                       "       2
 BUS3  EQU        8,F                       "       3
 CORE  EQU        9,F                 FACILITY FOR GETTING CORE STATS
*
*
*QUEUES:
*
 BUSQ0 EQU        1,Q                 BUS QUEUE 0
 BUSQ1 EQU        2,Q                 BUS QUEUE 1
 BUSQ2 EQU        3,Q                 BUS QUEUE 2
 BUSQ3 EQU        4,Q                 BUS QUEUE 3
*
*
*USER CHAIN:
*
 COMPQ EQU        1,C                 COMPLETION QUEUE
 RCMQ  EQU        2,C                 CHAN FOR ROM DELAYS
 MEINC EQU        3,C                 CHAIN FOR MEINC DELAY
 XDLQ0 EQU        4,C                 CHAIN FOR XFER DELAY
 XDLQ1 EQU        5,C                 CHAIN FOR XFER DELAYS-- BUS1
 XDLQ2 EQU        6,C                 CHAIN FOR XFER DELAYS-- BUS2
 XDLQ3 EQU        7,C                 CHAIN FOR XFER DELAYS-- BUS3
 SDLQ0 EQU        8,C                 CHAIN FOR SOURCE DELAYS -- BUS0
 SDLQ1 EQU        9,C                 CHAIN FOR SOURCE DELAYS -- BUS1
 SDLQ2 EQU        10,C                CHAIN FOR SOURCE DELAYS -- BUS2
 SDLQ3 EQU        11,C                CHAIN FOR SOURCE DELAYS -- BUS3
*
*
*LOGIC SWITCHES:
*
*                                                          INIT VALUE
 MAV0  EQU        1,L                 MODULE AVAILABLE 0       1
 MAV1  EQU        2,L                   "        "     1       1
 MAV2  EQU        3,L                   "        "     2       1
 MAV3  EQU        4,L                   "        "     3       1
```

```
 INSTF EQU       5,L                 INSTRUCTION FLAG                1  8r
 BMR0  EQU       6,L                 BUS MASTER FLAG 0               1
 BMR1  EQU       7,L                  "    "      "   1               1
 BMR2  EQU       8,L                  "    "      "   2               1
 BMR3  EQU       9,L                  "    "      "   3               1
 BHLD0 EQU       10,L                BUS HOLD FLAG 0                 1
 BHLD1 EQU       11,L                 "   "     "   1                 1
 BHLD2 EQU       12,L                 "   "     "   2                 1
 BHLD3 EQU       13,L                 "   "     "   3                 1
 BCC0  EQU       14,L                BUS CYCLE COMPLETE 0            1
 BCC1  EQU       15,L                 "     "        "     1          1
 BCC2  EQU       16,L                 "     "        "     2              1
 BCC3  EQU       17,L                 "     "        "     3              1
 ACK0  EQU       18,L                 SOURCE ACKNOWLEDGE BUS 0        0
 ACK1  EQU       19,L                   "          "       "  1        0
 ACK2  EQU       20,L                   "          "       "  2        0
 ACK3  EQU       21,L                   "          "       "  3        0
 BRQ10 EQU       22,L                 BUS REQUEST, DEVICE #1 BUS0     0
 BRQ11 EQU       23,L                  "      "        "     #1  " 1    0
 BRQ12 EQU       24,L                  "      "        "     #1  " 2    0
 BRQ13 EQU       25,L                  "      "        "     #1  " 3    0
 BRQ20 EQU       26,L                  "      "        "     #2  " 0    0
 BRQ21 EQU       27,L                  "      "        "     #2  " 1    0
 BRQ22 EQU       28,L                  "      "        "     #2  " 2    0
 BRQ23 EQU       29,L                  "      "        "     #2  " 3    0
 BRQ30 EQU       30,L                  "      "        "     #3  " 0    0
 BRQ31 EQU       31,L                  "      "        "     #3  " 1    0
 BRQ32 EQU       32,L                  "      "        "     #3  " 2    0
 BRQ33 EQU       33,L                  "      "        "     #3  " 3    0
 BGT10 EQU       34,L                BUS GRANT TO CEVICE #1, BUS 0 0
 BGT11 EQU       35,L                 "     "    "   "       "   " 1 0
 BGT12 EQU       36,L                 "     "    "   "       "   " 2 0
 BGT13 EQU       37,L                 "     "    "   "       "   " 3 0
 XEN0  EQU       38,L                TRANSFER ENABLE   BUS 0         0
 XEN1  EQU       39,L                   "         "       "  1         0
 XEN2  EQU       40,L                   "         "       "  2         0
 XEN3  EQU       41,L                   "         "       "  3         0
 MAV   EQU       42,L                CORE MEMORY IS AVAILABLE
 DAV   EQU       43,L                CORE MEMORY DATA IS AVAILABLE
 RCMEN EQU       44,L                MEMORY ENABLE FOR ROM'S
 ALUAV EQU       45,L                ALU IS AVAILABLE
 CCND0 EQU       46,L                CONDITION CODE #0
 COND1 EQU       47,L                CONDITION CODE #1
 COND2 EQU       48,L                CONDITION CODE #2
 COND3 EQU       49,L                CONDITION CODE #3
 CCND4 EQU       50,L                CONDITION CODE #4
*
*
*ARITHMETIC VARIABLES:  (VARIABLE NAME WITH "I" SUFFIX  IS AN ENTITY
*INDEX FUNCTION WHICH ADDS A BASE CONSTANT TO BUS NUMBER.)
*
 ADDI   VARIABLE    P3+K1               MODULE ADDRESS INDEX
 MAVI   VARIABLE    XH$MEN+K1           MODULE AVAILABLE INDEX
 MEINC  VARIABLE    (XH$MEN+K1)@4       MODULE ENABLE INCREMENT
 AMINC  VARIABLE    (XH$AMP+K1)@4       ACTIVE MODULE INCREMENT
 BMRI   VARIABLE    P4+K6               BUS MASTER INDEX
 BHLDI  VARIABLE    P4+K10              BUS HOLD INDEX
 BCCI   VARIABLE    P4+K14              BUS CYCLE COMPLETE INDEX
 BUSQI  VARIABLE    P4+K1               BUS QUEUE INDEX
 IMMED  VARIABLE    K1                  SOURCE CODE FOR IMMED DATA = 1
 BUSI   VARIABLE    P4+K5               BUS FACILITY INDEX
```

```
 IMRGI VARIABLE   XH$IMRP+K5              IMMEDIATE DATA REG SAVEV INDES
 ACKI  VARI       XH$IMRP+K18             ACK INDEX FOR ASSERTION OF IDAT
 XENI  VARIABLE   P4+K38                  TRANSFER ENABLE INDEX
 MADD  VARIABLE   P2-K1                   CCMPUTE MODULE ADDRESS
 PCINC VARIABLE   XH*2+1                  INCREMENT APPROPRIATE PC
 CCNDI VARI       P8+46                   INDEX TO COND-CODE LOGIC SWITCH
 POSTI VARI       XH$XBUS1/1000           THOUSANDS DIGIT = # OF POSTI'S
 ABNO  VARI       CH$CCMPQ-V$POSTI        NC. OF BUS CYCLES TO ABORT
 LSBS  VARI       (XH$XBUS1-V$POSTI)@4    LSB'S OF BRANCH ADDRESS
 MSBS  VARI       (XH$XBUS1-V$POSTI-V$LSBS)/4  MSB'S CF BRANCH ADDRESS
 XBUSI VARI       XH$IMRP+K13             INDEX TO WHICH XBUS TO ASSERT
 SDLQI VARI       P4+8                    SCURCE DELAY QUEUE INDEX
 XDLQI VARI       P4+4                    XFER DELAY QUEUE INDEX
 REGOI VARI       K25                     BASE ADDRESS OF BUS 0 REG SET
 REG1I VARI       K29                     BASE ADDRESS OF BUS 1 REG SET
 REG2I VARI       K33                     BASE ADDRESS OF BUS 2 REG SET
 REG3I VARI       K37                     BASE ADDRESS OF BUS 3 REG SET
 CORAC VARI       XH$MAR+60               INDEX TO GET CORE XH'S
*
*
*BOOLEAN VARIABLES:
*
 DELQ  BVARIABLE  (LS$BCC0*LS$XEN0)+(LS$BCC1*LS$XEN1)+BV$DELQC
 DELQC BVARIABLE  (LS$BCC2*LS$XEN2)+(LS$BCC3*LS$XEN3)    (CONTINUATICN)
*DELQ+DELQC ALLCWS THE FRCNT OF THE INSTRUCTION COMPLETION QUEUE TO BE
*DELETED AND FCR A TRANSFER ENABLE (XEN) TC BE ISSUED.
*
 MODI0 BVARIABLE  (XH$MEN'E'K0)*LS$MAV0   MODULE 0 IS READY TO GO
 MODI1 BVARIABLE  (XH$MEN'E'K1)*LS$MAV1   MODULE 1 IS READY TO GO
 MCDI2 BVARIABLE  (XH$MEN'E'K2)*LS$MAV2   MCDULE 2 IS READY TO GO
 MCDI3 BVARIABLE  (XH$MEN'E'K3)*LS$MAV3   MODULE 3 IS READY TO GO
 MINIT BVARIABLE  BV$MODI0+EV$MODI1+BV$MCDI2+BV$MODI3*LS$ROMEN
*                                                 SHOULD GO IF = 1
 BARBC BVARIABLE  LR$BGD0*(LS$BRQ10+LS$BRC20+LS$BRQ30)
 BARB1 BVARIABLE  LR$BGD1*(LS$BRQ11+LS$BRC21+LS$BRQ31)
 BARB2 BVARIABLE  LR$BGD2*(LS$BRQ12+LS$BRC22+LS$BRQ32)
 BARB3 BVARIABLE  LR$BGD3*(LS$BRQ13+LS$BRC23+LS$BRQ33)
*BARBI SIGNALS ACTIVATE BUS MASTER ARBITRATION CONTROLLERS (1 PER BUS)
*
*
*INITIALIZATICN CF LOGIC SWITCHES:
*
       INITIAL    LS1/LS2/LS3/LS4/LS5/LS6/LS7/LS8/LS9/LS10/LS11/LS12
       INITIAL    LS13/LS14/LS15/LS16/LS17
       INITIAL    LS42/LS43/LS44/LS45
*
*
*INITIALIZATICN CF CCNTROL STCRE:
*
*
*
*********************  MICRC PRCGRAM  ***************************
*
*      THE MICROPROGRAM IS STCREC IN 4 MATRIX SAVEVALUES WHICH REPRE-
*SENT THE FOUR RCM MODULES.  EACH CCLUMN REPRESENTS A MICRO INSTRUCTION.
*COLUMNS CORRESPOND TO ADDRESSES OFFSET BY ONE.  THE ROWS CORRESPOND TO
*MICRO INSTRUCTION FIELDS AS FCLLCWS:
*                                        RCW1 = BUS ADDRESS
*                                        RCW2 = SOURCE ADDRESS
*                                        RCW3 = SOURCE FUNCTION
*                                        RCW4 = DESTINATION ADDRESS
```

```
*                                        RCW5 = DESTINATION FUNCTION
*                                                                    88
*
*************************MICRO-DIAGNOSTIC #3*************************
*
        INITIAL     MH1(2,1),1/MH1(4,1),9
        INITIAL     MH2(3,1),20
        INITIAL     MH3(2,1),7/MH3(4,1),2
        INITIAL     MH4(2,1),1/MH4(4,1),9
        INITIAL     MH1(3,2),21
        INITIAL     MH2(2,2),7/MH2(4,2),6
        INITIAL     MH3(1,2),1/MH3(2,2),6/MH3(4,2),3
        INITIAL     MH4(1,2),1/MH4(2,2),1/MH4(4,2),8
        INITIAL     MH1(3,3),40
        INITIAL     MH2(2,3),1/MH2(4,3),9/MH2(5,3),1
        INITIAL     MH3(3,3),22
        INITIAL     MH4(2,3),7/MH4(4,3),8
        INITIAL     MH1(2,4),1/MH1(4,4),7/MH1(5,4),3
        INITIAL     MH2(3,4),2
        INITIAL     MH3(1,4),1/MH3(2,4),1/MH3(4,4),7/MH3(5,4),6
        INITIAL     MH4(3,4),20
        INITIAL     MH1(1,6),1/MH1(2,6),7/MH1(4,6),4
*
*
*
*************************INITIALIZATION OF CORE MEMORY*************************
*
        INITIAL     XH80,30                 (ADDRESS 20)
        INITIAL     XH81,31                 (ADDRESS 21)
        INITIAL     XH82,32                 (ADDRESS 22)
*
*
*
*
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                        SIMULATION MODEL BLOCKS                        *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
*
*LOC    OPERATION   A,B,C,D,E,F,G           COMMENTS
*
        GENERATE    ,,,,,8
        TEST E      BV$MINIT,K1             INITIATE A MEM CYCLE MAV(MEN)=1
        SAVEV       SERNO+,K1,H             SERIAL NUMBER FOR INST'S
        ASSIGN      1,XH$SERNO              ASSIGN SERIAL NO.
        ASSIGN      2,V$MAVI                P2=ROM FACILITY NO.
        ASSIGN      3,XH*2                  P3=MODULE PC
        ASSIGN      4,MH*2(1,V$ADDI)        P4=BUS ADDRESS (READ FROM ROM)
        ASSIGN      5,MH*2(2,V$ADDI)        P5=SOURCE ADDRESS
        ASSIGN      6,MH*2(3,V$ADDI)        P6=SOURCE FUNCTION
        ASSIGN      7,MH*2(4,V$ADDI)        P7=DEST ADD
        ASSIGN      8,MH*2(5,V$ADDI)        P8=DEST FUNCTION
        SPLIT       1,INCME                 SEND XACTION TO INC MOD ENABLE
        SEIZE       *2                      ACCESS PRESENTLY ENABLED ROM
        LOGIC R     *2                      RESET MAV OF ROM JUST ACCESSED
        SPLIT       1,RCMQU                 SEND OFFSPRING TO JOIN ROMQ
        LINK        ROMQ,FIFO
```

```
 ROMQU ADVANCE     100                    T-ACCESS = 100NS
       UNLINK      ROMQ,MICRC,1,1         UNLINK & SEND TO MICRO
       TERMINATE
 MICRC RELEASE     *2                     RELEASE ROM FACILITY
       TEST E      XH$AMP,V$MADD          LET XACTN ENABLED BY AMP TO GO
       GATE LS     INSTF,IDATA            IF INSTF = 0, GO TO IDATA
       QUEUE       V$BUSQI                GATHER STATS ON WAINING FOR BUS
 BUSOK GATE LS     V$BCCI                 WAIT FOR BUS CYCLE COMPLETE
       DEPART      V$BUSQI                LEAVE QUEUE
       TEST NE     P5,V$IMMED,IMMED       IF SOURCE IS IMMED, GO TO IMMED
 ISTB  SEIZE       V$BUSI                 SEIZE BUS FACILITY
       LOGIC S     *2                     SET MAV OF ROM JUST UNLOADED
       SAVEVALUE   *2,V$PCINC,H           INCREMENT MODULE PC
       SAVEVALUE   AMP,V$AMINC,H          INCREMENT ACTIVE MODULE POINTER
       LOGIC R     V$BCCI                 CLEAR BCC
       SPLIT       1,INSTQ                SEND ONE OFFSRING TO JOIN CCMPQ
       TEST NE     P4,K0,BSDC0            *
       TEST NE     P4,K1,BSDC1            *
       TEST NE     P4,K2,BSDC2            *    THE BUS ADDRESS IS DECODED
       TRANSFER    ,BSDC3                 GC TO BUS DECODER/CONTROLLER--3
*
*
*
 INCME SPLIT       1,STAGR                SEND OFFSPRING TO DELAY
       LINK        MEINC,FIFO             PUT IN DELAY QUEUE
 STAGR ADVANCE     10                     10 NS DELAY TO INC MOD ENABLE
       UNLINK      MEINC,INCIT,1,1        REMOVE FROM DELAY QUEUE
       TERMINATE
 INCIT SAVEV       MEN,V$MEINC,H          INCREMENT MODULE ENABLE
       BUFFER                             RESTART CURRENT EVENTS SCAN
       TERMINATE                          KILL IT
*
*
*
 IDATA SAVEVALUE   V$IMRGI,P6,H           LOAD IMMEDIATE DATA REG (NOTE
*                                          THAT P6 CONTAINS THE DATA)
       SAVEVALUE   V$XBUSI,P6,H           ALSO ASSERT IMMED DATA ON XBUS
       LOGIC S     V$ACKI                 ASSERT APPROPRIATE ACKNOWLEDGE
       LOGIC S     INSTF                  SET INST FLAG
       LOGIC S     *2                     SET MAV
       SAVEVALUE   *2,V$PCINC,H           INCREMENT MODULE PC
       SAVEVALUE   AMP,V$AMINC,H          INCREMENT AMP
       TERMINATE                          KILL THE TRANSACTION
*
*
*
*     THIS SEGMENT CREATES XACTIONS AS REQ'D TO POP THE COMPLETION QUEU
*
       GENERATE    ,,,,,0
       TEST E      BV$DELQ,1              DELETE TOP OF INSTQ?
       UNLINK      COMPQ,XEN,1
       LOGIC R     XEN0
       LOGIC R     XEN1                   CLEAR CURRENTLY SET XEN
       LOGIC R     XEN2                   (CNLY ONE SHOULD BE ON AT A TIM
       LOGIC R     XEN3
       TERMINATE
*
*
*   THIS SEGMENT IS ENTERED BY TRANSACTIONS WHICH HAVE JUST BEEN RE-
*MOVED FROM THE COMPLETION QUEUE WHICH IS STORED IN USER CHAIN "COMPQ"
*
```

```
 XEN    LOGIC S     V$XENI                  ASSERT XEN
        TERMINATE                           KILL IT
*
*
*       THIS SEGMENT IS ENTERED WHEN AN INST SPECIFIES IMMED DATA
*
 IMMED  SAVEVALUE   IMRP,P4,H               SAVE BUS ADD IN POINTER REG
*                                           SO THAT THE DATA KNOWS WHERE TO
*                                           BE LOADED
        LOGIC R     INSTF                   CLEAR INSTF- NEXT WORD IS DATA
        TRANSFER    ,ISTB                   GO TO ISTB
*
*
*
*THIS SEGMENT IS ENTERED BY AN OFFSPRING OF EVERY INST.  IT PLACES THE
*INSTRUCTION IN THE INSTRUCTION QUEUE
*
 INSTQ  LINK        COMPQ,FIFO,XEN          PUT INST IN COMP QUEUE UNLESS
        TERMINATE                           IT IS THE VERY FIRST INSTRUCTION
*
*
*
 BSDCO  SPLIT       1,SDCO
        GATE LS     ACKO                    WAIT FOR ACKNOXLEDGE FROM SOURCE
        SPLIT       1,XDLO                  *
        LINK        XDLQO,FIFO              *
 XDLO   ADVANCE     20                      *20NS INTERRUPTABLE DELAY
        UNLINK      XDLQO,XGOO,1            *
        TERMINATE                           *
 XGOO   GATE LS     XENO                    WAIT FOR X-ENABLE FOR XFER
        TEST NE     P7,K1,IMRDO             *
        TEST NE     P7,K2,RRDO              *
        TEST NE     P7,K3,RRDO              *
        TEST NE     P7,K4,RRDO              *
        TEST NE     P7,K5,RRDO              *
        TEST NE     P7,K6,UCUDO             *
        TEST NE     P7,K7,ALL               *
        TEST NE     P7,K8,ALR               *
        TEST NE     P7,K9,MEMAR             *
        SAVEVALUE   ILLDO,K1,F              SET DEST ERROR BUSO FLAG
 FINDO  LOGIC R     ACKO                    RETURN POINT FROM DEV SEGMENTS
        ADVANCE     10                      10NS NONINTERRUPTABLE DELAY
        LOGIC S     BCCO                    SET BUS CYCLE COMPLETE
        RELEASE     V$BUSI                  RELEASE BUS FACILITY
        PRINT       ,,C,X
        PRINT       ,,MOV,X
        PRINT       ,,XH,X
        PRINT       ,,LG,X
        TERMINATE
*
*
 SDCO   TEST NE     P5,K1,BUKET             IF IMMEDIATE INST --DISCARD
        SPLIT       1,SDLO                  *
        LINK        SDLQO,FIFO              *
 SDLO   ADVANCE     50                      *INTERRUPTABLE 50NS DELAY
        UNLINK      SDLQO,AKGCO,1,1         *
        TERMINATE                           *
 AKGOO  TEST NE     P5,K1,IMRSO             **
        TEST NE     P5,K2,RRSC              **
        TEST NE     P5,K3,RRSC              **
        TEST NE     P5,K4,RRSC              **
```

```
        TEST NE     P5,K5,RRSO          **
        TEST NE     P5,K6,UCUSO         **
        TEST NE     P5,K7,MEMCR         **
        SAVEVALUE   ILLSO,K1,H          ERROR IF THIS BLOCK IS ENTERED
 FINSO  LOGIC S     ACKO                RETURN POINT FROM SOURCE SEG.
        TERMINATE                       KILL IT
*
*
*
*
 BSDC1  SPLIT       1,SDC1
        GATE LS     ACK1                WAIT FOR ACKNOXLEDGE FROM SOURCE
        SPLIT       1,XDL1              *
        LINK        XDLQ1,FIFO          *
 XDL1   ADVANCE     20                  *20NS INTERRUPTABLE DELAY
        UNLINK      XDLQ1,XGO1,1        *
        TERMINATE                       *
 XGO1   GATE LS     XEN1                WAIT FOR X-ENABLE FOR XFER
        TEST NE     P7,K1,IMRC1         *
        TEST NE     P7,K2,RRC1          *
        TEST NE     P7,K3,RRD1          *
        TEST NE     P7,K4,RRD1          *
        TEST NE     P7,K5,RRD1          *
        TEST NE     P7,K6,UCUC1         *
        TEST NE     P7,K7,BCU           *
        TEST NE     P7,K8,MEMIR         *
        SAVEVALUE   ILLD1,K1,H          SET DEST ERROR BUS1 FLAG
 FIND1  LOGIC R     ACK1                RETURN POINT FROM DEV SEGMENTS
        ADVANCE     10                  10NS NONINTERRUPTABLE DELAY
        LOGIC S     BCC1                SET BUS CYCLE COMPLETE
        RELEASE     V$BUS1              RELEASE BUS FACILITY
        PRINT       ,,MOV,X
        PRINT       ,,C,X
        PRINT       ,,LG,X
        PRINT       ,,XH,X
        TERMINATE   1
*
*
 SDC1   TEST NE     P5,K1,BUKET         IF IMMEDIATE INST --DISCARD
        SPLIT       1,SDL1              *
        LINK        SDLQ1,FIFO          *
 SDL1   ADVANCE     50                  *INTERRUPTABLE 50NS DELAY
        UNLINK      SDLQ1,AKGC1,1,1     *
        TERMINATE                       *
 AKGO1  TEST NE     P5,K1,IMRS1         **
        TEST NE     P5,K2,RRS1          **
        TEST NE     P5,K3,RRS1          **
        TEST NE     P5,K4,RRS1          **
        TEST NE     P5,K5,RRS1          **
        TEST NE     P5,K6,UCLS1         **
        TEST NE     P5,K7,ALO           **
        SAVEVALUE   ILLS1,K1,H          ERROR IF THIS BLOCK IS ENTERED
 FINS1  LOGIC S     ACK1                RETURN POINT FROM SOURCE SEG.
        TERMINATE                       KILL IT
*
*
*
*
 BSDC2  SPLIT       1,SDC2
        GATE LS     ACK2                WAIT FOR ACKNOXLEDGE FROM SOURCE
        SPLIT       1,XDL2              *
```

```
        LINK        XDLQ2,FIFC          *
XDL2    ADVANCE     20                  *20NS INTERRUPTABLE DELAY
        UNLINK      XDLQ2,XGO2,1        *
        TERMINATE                       *
XGO2    GATE LS     XEN2                WAIT FOR X-ENABLE FOR XFER
        SAVEVALUE   ILLD2,K1,H          SET DEST ERROR BUS2 FLAG
FIND2   LOGIC R     ACK2                RETURN POINT FROM DEV SEGMENTS
        ADVANCE     10                  10NS NONINTERRUPTABLE DELAY
        LOGIC S     BCC2                SET BUS CYCLE COMPLETE
        RELEASE     V$BUSI              RELEASE BUS FACILITY
        TERMINATE                       KILL IT
*
*
SDC2    TEST NE     P5,K1,BUKET         IF IMMEDIATE INST --DISCARD
        SPLIT       1,SDL2              *
        LINK        SDLQ2,FIFC          *
SDL2    ADVANCE     50                  *INTERRUPTABLE 50NS DELAY
        UNLINK      SDLQ2,AKGC2,1,1     *
        TERMINATE                       *
AKGO2   SAVEVALUE   ILLS2,K1,H          ERROR IF THIS BLOCK IS ENTERED
FINS2   LOGIC S     ACK2                RETURN POINT FROM SOURCE SEG.
        TERMINATE                       KILL IT
*
*
*
*
BSDC3   SPLIT       1,SDC3
        GATE LS     ACK3                WAIT FOR ACKNOXLEDGE FROM SOURCE
        SPLIT       1,XDL3              *
        LINK        XDLQ3,FIFC          *
XDL3    ADVANCE     20                  *20NS INTERRUPTABLE DELAY
        UNLINK      XDLQ3,XGO3,1        *
        TERMINATE                       *
XGO3    GATE LS     XEN3                WAIT FOR X-ENABLE FOR XFER
        SAVEVALUE   ILLD3,K1,H          SET DEST ERROR BUS3 FLAG
FIND3   LOGIC R     ACK3                RETURN POINT FROM DEV SEGMENTS
        ADVANCE     10                  10NS NONINTERRUPTABLE DELAY
        LOGIC S     BCC3                SET BUS CYCLE COMPLETE
        RELEASE     V$BUSI              RELEASE BUS FACILITY
        TERMINATE                       KILL IT
*
*
SDC3    TEST NE     P5,K1,BUKET         IF IMMEDIATE INST --DISCARD
        SPLIT       1,SDL3              *
        LINK        SDLQ3,FIFC          *
SDL3    ADVANCE     50                  *INTERRUPTABLE 50NS DELAY
        UNLINK      SDLQ3,AKGC3,1,1     *
        TERMINATE                       *
AKGO3   SAVEVALUE   ILLS3,K1,H          ERROR IF THIS BLOCK IS ENTERED
FINS3   LOGIC S     ACK3                RETURN POINT FROM SOURCE SEG.
        TERMINATE                       KILL IT
*
*
*
*
RRS0    ASSIGN      5+,V$REGOI          ADD BASE TO P5=RELATIVE ADDRESS
        SAVEVALUE   XBUS0,XH*5,H        ASSERT $BUS
        TRANSFER    ,FINS0              QUIT
*
RRD0    ASSIGN      7+,V$REGOI          ADD BASE TO P7=REL ADDRESS
        SAVEVALUE   *7,XH$XBUS0,H       ASSERT REG CONTENTS ON X-BUS
```

```
        TRANSFER    ,FINDO
*
*   UNIVERSAL COMMUNICATIONS UNIT
 UCUDO  SAVEVALUE   UCOMR,XH$XBUSO,H      LOAD UCOMREG W/ (XBUS)
        TRANSFER    ,FINDO               RETURN TO BUS DECODE/CONTROL
*
*
 UCUSO  GATE LS     XENO                 NEED TO WAIT TIL PREVIOUS CYCLE
        SAVEVALUE   XBUSO,XH$UCOMR,H      LOAD (UCOMR) ONTO XBUS
        TRANSFER    ,FINSO               RETURN TO BUS DECODE/CONTROL
*
*
*
 RRS1   ASSIGN      5+,V$REG1I           ADD BASE TO P5=RELATIVE ADDRESS
        SAVEVALUE   XBUS1,XH*5,H         ASSERT $BUS
        TRANSFER    ,FINS1               QUIT
*
 RRD1   ASSIGN      7+,V$REG1I           ADD BASE TO P7=REL ADDRESS
        SAVEVALUE   *7,XH$XBUS1,H        ASSERT REG CONTENTS ON X-BUS
        TRANSFER    ,FIND1
*
*   UNIVERSAL COMMUNICATIONS UNIT
 UCUD1  SAVEVALUE   UCOMR,XH$XBUS1,H      LOAD UCOMREG W/ (XBUS)
        TRANSFER    ,FIND1               RETURN TO BUS DECODE/CONTROL
*
*
 UCUS1  GATE LS     XEN1                 NEED TO WAIT TIL PREVIOUS CYCLE
        SAVEVALUE   XBUS1,XH$UCOMR,H      LOAD (UCOMR) ONTO XBUS
        TRANSFER    ,FINS1               RETURN TO BUS DECODE/CONTROL
*
*
*
*
 RRS2   ASSIGN      5+,V$REG2I           ADD BASE TO P5=RELATIVE ADDRESS
        SAVEVALUE   XBUS2,XH*5,H         ASSERT $BUS
        TRANSFER    ,FINS2               QUIT
*
 RRD2   ASSIGN      7+,V$REG2I           ADD BASE TO P7=REL ADDRESS
        SAVEVALUE   *7,XH$XBUS2,H        ASSERT REG CONTENTS ON X-BUS
        TRANSFER    ,FIND2
*
*   UNIVERSAL COMMUNICATIONS UNIT
 UCUD2  SAVEVALUE   UCOMR,XH$XBUS2,H      LOAD UCOMREG W/ (XBUS)
        TRANSFER    ,FIND2               RETURN TO BUS DECODE/CONTROL
*
*
 UCUS2  GATE LS     XEN2                 NEED TO WAIT TIL PREVIOUS CYCLE
        SAVEVALUE   XBUS2,XH$UCOMR,H      LOAD (UCOMR) ONTO XBUS
        TRANSFER    ,FINS2               RETURN TO BUS DECODE/CONTROL
*
*
*
 RRS3   ASSIGN      5+,V$REG3I           ADD BASE TO P5=RELATIVE ADDRESS
        SAVEVALUE   XBUS3,XH*5,H         ASSERT $BUS
        TRANSFER    ,FINS3               QUIT
*
 RRD3   ASSIGN      7+,V$REG3I           ADD BASE TO P7=REL ADDRESS
        SAVEVALUE   *7,XH$XBUS3,H        ASSERT REG CONTENTS ON X-BUS
        TRANSFER    ,FIND3
*
*   UNIVERSAL COMMUNICATIONS UNIT
```

```
 UCUD3 SAVEVALUE  UCOMR,XH$XBUS3,H      LOAD UCOMREG W/ (XBUS)
       TRANSFER   ,FIND3                RETURN TO BUS DECODE/CONTROL
*
*
 UCUS3 GATE LS    XEN3                  NEED TO WAIT TIL PREVIOUS CYCLE
       SAVEVALUE  XBUS3,XH$UCOMR,H      LOAD (UCOMR) ONTO XBUS
       TRANSFER   ,FINS3                RETURN TO BUS DECODE/CONTROL
*
*
*
*     THIS SEGMENT IS ENTERED FROM A BUS DECODER. IT SIMULATES THE
*ACTIONS OF THE BRANCH CONTROL UNIT.  THE BRANCH ALGORITHM ENVOLVES
*TESTING THE LENGTH OF THE COMPLETION QUEUE (COMPQ),  ABORTING
*APPROPRIATE MEMORY AND BUS CYCLES, AND ALTERING THE PROGRAM COUNTER
*WHEN A BRANCH IS SUCCESSFUL.
 BCU   TEST LE    P8,K5,BITCL            BIT SET OR BIT CLEAR CONDITION?
       GATE LS    V$CONDI,BCONE          BRANCH FAILS--GO TO BDONE
       TRANSFER   ,JUMP                  BRANCH IS SUCCESSFUL
 BITCL ASSIGN     8-,K5                  SUBTRACT TO GET SWITCH INDEX
       GATE LR    V$CONDI,BCONE          BRANCH FAILS--GO TO BDONE
 JUMP  TEST GE    CH$COMPQ,V$PCSTI       CHECK # OF LOADED INSTS, &WAIT
*                                        IF NECESSARY
       LOGIC R    ROMEN                  DISABLE NEW ROM    INITIATES.
       UNLINK ROMQ,ABMEM,ALL             ABORT ALL MEM CYCLES IN PROGRESS
 UNLINK           COMPQ,ABBUS,V$ABNO,BACK   ABORT BAD BUS CYCLES
       SAVEVALUE  AMP,V$LSBS,H           *
       SAVEVALUE  MEM,XH$AMP,H           *
       SAVEVALUE  MAR0,V$MSES,H          *LOAD NEW PROGRAM COUNTER
       SAVEVALUE  MAR1,XH$MAR0,H         *
       SAVEVALUE  MAR2,XH$MAR0,H         *
       SAVEVALUE  MAR3,XH$MAR0,H         *
       DEPART     1,Q1                   *
       DEPART     2,Q2                   *ZERO ALL BUSS QUEUES--NOTHING
       DEPART     3,Q3                   *SHOULD BE WAITING FOR A BCC
       DEPART     4,Q4                   *AT THIS TIME.
       TRANSFER   ,BDONE                 GO TO 'FINISH
 ABMEM LOGIC S    *2                     SET MAV OF ABORTED MEMORY CYCLES
       RELEASE    *2                     RELEASE ROM FACILITY OF ABBORT
 BUKET TERMINATE                         THIS IS THE BIT BUCKET
 ABBUS UNLINK     V$SDLQI,BUKET,1        *THROW ALL X'S WAITING IN DELAYS
       UNLINK     V$XDLQI,BUKET,1        *IN THE BIT BUCKET FOR ALL BAD
*                                        *BUS CYCLES STARTED.
       LOGIC R    V$ACKI                 RESET ACK
 BDONE LOGIC S    ROMEN                  ENABLE ROM'S   AGAIN
       TRANSFER   ,FIND1
*
*
*
*
*   THIS SEGMENT SIMULATES CORE MEMORY.  SECTIONS FOR MAR,MDOR,MDIR ARE
* INCLUDED.  MEMORY ACCESS IS INTERLOCKED BY MAV ON ADDRESS AND INPUT
* DATA AS DESTINATION, AND BY CAV ON OUTPUT DATA AS A SOURCE
 MEMIR GATE LS    MAV                    WAIT FOR MEMORY AVAILABLE
       SAVEVALUE  MDIR,XH$XBUS1,H        LOAD IR FROM X-BUS
       TRANSFER   ,FIND1                 RETURN TO BUS DECODE/CONTROL
 MEMOR GATE LS    CAV                    WAIT FOR MAV=1
       SAVEVALUE  XBUS0,XH$MDOR,H        ASSERT X-BUS FROM OD REG
       TRANSFER   ,FINS0                 RETURN TO BUS CONTROLLER
 MEMAR GATE LS    MAV                    WAIT FOR MAV=1
       SAVEVALUE  MAR,XH$XBUS0,H
       SPLIT      1,MCCNT                SEND OFFSRING TO DO MEM CONTROL
```

```
        TRANSFER    ,FINDO
MCONT   SEIZE       CORE                    TAKE FAX. STATS ON CORE
        LOGIC R     CAV                     CLEAR DAV
        LOGIC R     MAV                     CLEAR MAV
        ADVANCE     300                     T-ACCESS = 300NS
        ASSIGN      7,V$CORAC               P8=$AVEVALUE INDEX FOR CORE WORD
        SAVEVALUE   MDOR,XH*7,H             LOAD MDOR
        LOGIC S     DAV                     SET DAV
        ADVANCE     600                     T-RESTORE/WRITE = 600NS
        TEST E      P8,K1,RESTR             P8=1 IS MEMORY WRITE,ELSE READ
        SAVEVALUE   *7,XH$MDIF,H            DO MEMORY WRITE
RESTR   LOGIC S     MAV                     SET MAV
        RELEASE     CORE                    RELEASE
        TERMINATE                           KILL CONTROL TRANSACTION
*
*
*
*
*     THIS SEGMENT SIMULATES THE OPERATION OF THE ALU.  THE ACTUAL
*OPERATIONS ARE NOT DONE.  PROVISIONS ARE MADE FOR SETTING AND
*CLEARING OF VARIOUS CONDITION CODES WHICH WOULD BE AUTOMATICALLY SET
*IF ACTUAL OPERATIONS WERE SIMULATED.  LOADING OF ALL CAUSES EXECUTION
*OF THE  SPECIFIED OPERATION.  INTERLOCKING PREVENTS ACCESS OF INVALID
*RESULTS.  THE ACTUAL ARITH & LOG OPERATIONS MAY BE SIMULATED BY
*DECPDOMG TJE FIMCTOPN, AND BRANCHING TO 'HELP' BLOCKS.
*
*
*
ALL     GATE LS     ALUAV                   WAIT UNTIL ALU AVAILABLE
        SPLIT       1,ALUGO                 SEND OFFSPRING TO ALU CONTROLLE
        TRANSFER    ,FINDO                  COMPLETE BUS CYCLE
ALUGC   LOGIC R     ALUAV                   CLEAR ALU AVAILABLE
        TEST GE     P8,K10,LOGOP            P8<10 ARE LOGIC OPERATIONS
ARTOP   ADVANCE     160                     ARITH OPS TAKE 160NS
        TRANSFER    ,OPDON
LOGOP   ADVANCE     130                     LOGIC OPS TAKE 130NS
OPDON   LOGIC S     ALUAV                   OUTPUT VALID--SET ALU AVAILABLE
        TERMINATE                           OVER AND OUT
*
*
ALR     GATE LS     ALUAV                   WAIT UNTIL ALUAV = 1
        TRANSFER    ,FINDO                  GO BACK TO BUS CONTROOER
*
*
ALO     GATE LS     ALUAV                   WAIT TILL ALUAV = 1
        TRANSFER    ,FINS1
*
*
*
IMRS0   SAVEVALUE   XBUS0,XH$IMR0,H         ASSERT XBUS
        TRANSFER    ,FINS0
IMRD0   SAVEVALUE   IMR0,XH$XBUS0,H         LOAD EMREG FROM XBUS
        TRANSFER    ,FINDO                  RETURN TO DECODE/CONTROL
IMRS1   SAVEVALUE   XBUS1,XH$IMR1,H         ASSERT XBUS
        TRANSFER    ,FINS1                  RETURN TO BUS DECODE/CONTROL
IMRD1   SAVEVALUE   IMR1,XH$XBUS1,H         LOAD EMREG FROM XBUS
        TRANSFER    ,FIND1                  RETURN TO DECODE/CONTROL
IMRS2   SAVEVALUE   XBUS2,XH$IMR2,H         ASSERT XBUS
        TRANSFER    ,FINS2                  RETURN TO BUS DECODE/CONTROL
IMRD2   SAVEVALUE   IMR2,XH$XBUS2,H         LOAD EMREG FROM XBUS
        TRANSFER    ,FIND2                  RETURN TO DECODE/CONTROL
```

```
IMRS3  SAVEVALUE   XBUS3,XH$IMR3,H        ASSERT XBUS
       TRANSFER    ,FINS3                 RETURN TO BUS DECODE/CONTROL
IMRD3  SAVEVALUE   IMR3,XH$XBUS3,H        LOAD EMREG FROM XBUS
       TRANSFER    ,FIND3                 RETURN TO DECODE/CONTROL
*
*
       START       4
*
       END
```

APPENDIX C

IBM SYSTEM/360 EMULATOR MICROPROGRAM

The following listing is the partial system/360 emulator described in chapter 4. The symbolic microcode conventions described in chapters 2 and 4 apply. Literal source data is enclosed in single quotation marks, and is given in 4-digit hexadecimal if it is a numerical constant, and is given in its symbolic form if it is an address constant. Extensive comments, and the explanation given in chapter 4 should make the code easy to understand.

Label	Bus	Source	Destination	Function	Comment
*					"INIT" initiates the fetch of the
*					first half word of the program.
INIT	2	ICHI	CARH		Load high-order byte of instruction
*					counter into core address register.
	3	ICLO	CARL		Load low-order 16 bits of instruc-
*					tion counter into core address
*					register.
	3	ICLO	ALL	INC	Increment instruction counter.
	3	ALO	ICLO		Store updated instruction counter.
	0	'IFCH'	BCU	CCL	If carry clear then instruction
*					counter is ok. Go to "IFCH."
	2	ICHI	ALL	INC	Carry is set. Increment high-order
*					byte.
	2	ALO	ICHI		Store updated high-order byte of
*					instruction counter.
*					
*					
*					
*					"IFCH" gets the first half-word of
*					the program from core memory, and
*					decodes the first 4 bits, branching
*					to one of sixteen routines for final
*					decoding.
*					External interrupts are checked
*					first.
IFCH	1	'00FF'	ALR		Check low-order byte of STR for
*					pending interrupts.
	2	'STR'	ALL	AND	
	0	'EXT'	BCU	ZCL	"EXT" handles external interrupts
*					(unimplemented).
	8	ICLO	UCOM		Route ICLO to bus 0.
	1	CDR	IRO		Get 1st half word of current
*					instruction.
	0	UCOM	ALL	INC	
	2	ICHI	CARH		
	3	ICLO	CARL	R	Access next half word of program.
	0	'A'	BCU/1	CSET	If carry is set, increment high-
*					order byte of IC.
B	1	IRO	SHR	L4/EA	Shift high order 4 bits to low order
*					position.
	3	'000F'	ALR		Load mask into ALU.
	0	SHR	ALL	AND	Mark off all but the low 4 bits.
	2	ALO	SHR	L1/0	Double the format/type field to get
*					branch-table displacement.
	1	'DCBS'	ALR		Load the base address of the branch
*					table into ALU.
	0	SHR	ALL	ADD2	Add the displacement to get branch
*					table entry address.
	0	ALO	BCU	UC	Enter the branch table.

Label	Bus	Source	Destination	Function	Comment
A	2	ICHI	ALL	INC	Carry set, increment high byte of
*					instruction counter.
	0	'B'	BCU(1)	UC	Jump back to "B" after 1 post
*					instruction.
	2	ALO	ICHI		(Post instruction) Save updated
*					high-order byte of counter
*					instruction.
DCBS	0	'RRO'	BCU	UC	Branch table routes control to 1
*					of 16 routines.
	0	'RR1'	BCU	UC	
	0	'RR2'	BCU	UC	
	0	'RR3'	BCU	UC	
	0	RX0	BCU	UC	
	0	RX1	BCU	UC	
	0	RX2	BCU	UC	
	0	RX3	BCU	UC	
	0	RSI0	BCU	UC	
	0	RSI1	BCU	UC	
	0	RSI2	BCU	UC	
	0	RSI3	BCU	UC	
	0	'OPER4'	BCU	UC	Op-code error (4 byte instruction).
	0	'OPER4'	BCU	UC	Op-code error (4 byte instruction).
	0	'OPER6'	BCU	UC	Op-code error (6 byte instruction).
	0	'SS1'	BUC	UC	
	0	'OPER6'	BCU	UC	Op-code error (6 byte instruction).
	0	'SS3'	BCU	UC	
*					
*					
*					
*					"RRO" decodes bits <4-7> of the
*					current instruction.
*					This is done by testing the field
*					directly.
RRO	1	IRO	SHR	R8	Move instruction code bits to least
*					significant position.
		'000F'	ALR		Load mask into ALU.
	0	SHR	ALL	AND	Mask off all but instruction code
*					bits.
	1	ALO	ALR		Load instruction code into ALU.
	2	'0007'	ALL	XOR	Compare code to the "BCR" code.
	0	'BCR'	BCU	ZSET	If match jump to "BCU." Continue
*					similarly.
	2	'0005'	ALL	XOR	
	0	'BALR'	BCU	ZSET	
	2	'0006'	ALL	XOR	
	0	'BCTR'	BCU	ZSET	
	2	'0004'	ALL	XOR	
	0	'SPM'	BCU	ZSET	
	2	'0008'	ALL	XOR	
	0	'SSK'	BCU	ZSET	
	2	'0009'	ALL	XOR	

Label	Bus	Source	Destination	Function	Comment
	0	'SVC'	BCU	ZSET	
	0	'OPER2'	BCU	UC	Op-code error (2 byte instruction).
*					
*					
*					
*					"RR1" decodes the instruction code
*					(<4-7>) via a branch table, and
*					transfers control to execution
*					routines.
RR1	1	IR0	ALR		Load instruction into ALU.
	2	'000F'	ALL	AND	Load mask and zero all but R2 field.
	3	ALO	SHR	L1/0	Double R2 to get scratch memory
*					address.
	1	SHR	SAR	R	Access low order 16 bits of operand
*					two.
	2	'OF00'	ALL	AND	Mask all but instruction code field
*					from instruction.
	0	SODR	OP2LO		Get low-order half word of operand
*					two and save.
	3	ALO	SHR	R8/0	Right justify instruction code field.
	1	SHR	SHR	L1/0	Double this to get branch table
*					displacement.
	0	SODR	OP2HI		Get high-order half word of operand
*					2 and save.
	3	'RR1B'	ALR		Load base address of RR1 branch
*					table.
	2	SHR	ALL	ADD2	Add the displacement to get branch
*					table entry address.
	0	ALO	BCU	UC	Enter RR1 branch table.
RR1B	0	'LP0	BCU	UC	RR1 branch table starts here (2
*					words per entry).
		.			Unimplemented instructions.
		.			
		.			
		.			
	0	'LOAD'	BCU	UC	
	0	'CMP'	BCU	UC	
	0	'ADD'	BCU	UC	
	0	'SUB'	BCU	UC	
	0	'MR'	BCU	UC	
		:			Unimplemented instructions.
		:			
	0	'SLR'	BCU	UC	
*					
*					
*					
*					"RX0" does final decode for the
*					RX0 group of instructions. A
*					branch table with 4 words per entry
*					is used, which allows the execution
*					routine entry point to be stored
*					prior to the jump to the setup
*					routine.

Label	Bus	Source	Destination	Function	Comment
RX0	1	IR0	SHR	R8	Right justify instruction code field
*					(<4-7>) of current instruction.
	3	'000F'	ALR		Load mask into ALU.
	0	SHR	ALL	AND	And zero all but instruction code
*					bits.
	1	CDR	IR1		While waiting for ALU, get second
*					half word of current instruction.
	2	ALO	SHR	L2/0	Multiply instruction code bits by
*					4 to get branch table displacement.
	3	'RX0B'	ALR		Load base address of RX0 branch
*					table into ALU.
	2	SHR	ALL	ADD2	And add the displacement to get
*					table entry address.
	0	ALO	BCU	UC	Enter table.
RX0B	0	'STH'	BCU	UC	RX0 branch table starts here.
		Ø	Ø	Ø	Don't core words.
		Ø	Ø	Ø	
					Unimplemented instruction, skip
					to address = RX0B+20.
B+20	0	'BAL'	BCU	UC	
					Unimplemented instruction. Skip to
					address = RX0B+28.
B+28	0	'BC'	BCU	UC	
		Ø	Ø	Ø	Don't care words.
		Ø	Ø	Ø	
	0	'HWSU'	BCU/1	UC	(1 post instruction) Jump to half-
*					word setup routine.
	3	'LOAD'	LKRG		After saving entry address of
*					execution routine.
	0	'HWSU'	BCU/1	UC	
	3	'CMP'	LKRG		
	0	'HWSU'	BCU/1	UC	
	3	'ADD'	LKRG		
	0	'HWSU'	BCU/1	UC	
	3	'SUB'	LKRG		
					Unimplemented instructions.
*					
*					
*					
*					"RX1" uses subroutine "EA2" to
*					prepare the effective address of
*					operand 2. Final decode is then
*					accomplished via a 4 word/entry
*					branch table using "FWSU" to
*					complete the setup.
RX1	0	'EA2'	BCU/2	UC	Link to subroutine "EA2", with 2
*					post instructions.
	3	'C'	LKRG		(Post instruction) Save return
*					address.

```
Label   Bus   Source     Destination   Function   Comment

        1       CDR          IR1                 (Post instruction) Get second half-
*                                                 word of current instruction.
C       1       IRO          SHR         R8      Right-justify instruction code field
*                                                 (<4-7>).
        3     '000F'         ALR                 Load mask.
        2       SHR          ALL         AND     And all but instruction code bits.
        0       ALO          SHR         L2/0    Multiply by 4 to get branch table
*                                                 displacement.
        1     'RX1B'         ALR                 Load branch table base address.
        2       SHR          ALL         ADD2    Add the displacement to form table
*                                                 entry address.
        0       ALO          BCU         UC      Branch to table.
RX1B    0      'ST'          BCU         UC      (Branch table starts here.) Jump
*                                                 to "ST" to execute ST.
        0     'OPER4'        BCU         UC      Op-code error.
        0     'OPER4'        BCU         UC
        0     'OPER4'        BCU         UC
                 :                                }
                 :                                } Unimplemented instructions. Skip
*                                                 } to address = RXIB+32.
B+32    0      'FWSU'      BCU/1          UC      Jump to full word setup, "FWSU"
*                                                 after 1 post instruction.
        3      'LOAD'       LKRG                (Post instruction) Save execution
*                                                 routine for L.
        0      'FWSU'      BCU/1          UC
        3       'CMP'       LKRG
        0      'FWSU'      BCU/1          UC
        3       'ADD'       LKRG
        0      'FWSU'      BCU/1          UC
        3       'SUB'       LKRG
                 :                                }
                 :                                } Unimplemented instructions.
*                                                 }
*
*
*                                                 "BALR" sets up the BALR instruction,
*                                                 then jumps to "LINK."
BALR    1       IRO          ALR                 Load first half-word of instruction
*                                                 into ALU.
        0     '000F'         ALL         AND     Zero all but R2 field.
        3     '4000'         T1                  Save instruction length code.
        2       ALO          SHR         L1/0    Double R2 field to get scratch
*                                                 address.
        1       SHR          SAR         R       Access 1st half-word of operand 2.
        2      SODR        OP2LO                 Get 1st half-word of operand 2 and
*                                                 save.
        0     'LINK'       BCU/1          UC      Jump to "LINK" (After 1 post
*                                                 instruction) to complete BALR.
        2      SODR        OP2HI                 (Post instruction) Get and save 2
*                                                 half-word of operand 2.
```

Label	Bus	Source	Destination	Function	Comments
*					
*					
*					
*					"BCR" uses subroutine "TEST" to
*					check branch condition. If
*					unsuccessful, control returns to
*					"IFCH." If successful, the branch
*					address is gotten and subroutine
*					"JUMP" entered.
BCR	0	'TEST'	BCU/1	UC	Jump to "TEST." (1 post instruction)
	3	'41'	LKRG		(Post instruction) Save return
*					address.
U1	0	'IFCH'	BCU	UC	If "TEST" unsuccessful, return here,
*					and go to "IFCH".
S1	1	IRO	ALR		Load 1st half-word of instruction.
	2	'000F'	ALL	AND	Mask all but R2 field.
	3	ALO	SHR	L1/0	Double to get scratch memory
*					address.
	1	SHR	SAR	R	Access operand 2 is scratch memory.
	2	SODR	OP2LO		Get and save 1st half-word.
	0	'JUMP'	BCU/1	UC	Jump (after 1 post instruction) to
*					"JUMP."
	2	SODR	OP2HI		(Post instruction) Get and save 2nd
*					half-word.
*					
*					
*					
*					"BAL" uses subroutine "EA2" to
*					complete the effective address of
*					operand 2, and fetches operand 2,
*					the branch address. Control then
*					jumps to "LINK" to complete the
*					"BAL" instruction.
BAL	0	'EA2'	BCU/1		Link to "EA2."
	3	'D'	LKRG		(Post instruction)
D	2	EAHI	CARH		
	3	EALO	CARL	R	Access first half of branch address.
	3	EALO	UCOM		Route low order half word of
*					effective address.
	1	ALO	EALO		Store updated low-order half-word.
	0	'E'	BCU	CCL	Jump to "E" if high-order byte of
*					effective address is ok.
	2	EAHI	ALL	INC	Else increment high-order byte.
	2	ALO	EAHI		And store updated high-order byte.
E	1	CDR	UCOM		Route the 1st word of the branch
*					address to bus 2,
	2	UCOM	OP2LO		and save in OP2LO.
	2	EAHI	CARH		Access second word of branch address.
	3	EALO	CARL	R	

Label	Bus	Source	Destination	Function	Comment
	0	'LINK'	BCU/3	UC	Go to "LINK" to complete the BAL instruction (3 post instruction).
	3	'8000'	T1		Save instruction length code.
	1	CDR	UCOM		Route second word of branch address to bus 2.
	2	UCOM	OP2HI		Save second word of branch address.
*					
*					
*					
*					"BC" uses subroutine "TEST" to check condition.
*					If successful, "EA2" computes the effective address of the branch address, and "JUMP" completes the instruction. If unsuccessful, the next program half-word is accessed, and control returns to "IFCH."
BC	0	'TEST'	BCU/1		Go to "TEST" to check branch condition.
*					
	3	'UZ'	LKRG		(Post instruction) Save "unsuccessful" return address.
*					
U2	0	'UN'	BCU	UC	If unsuccessful, "TEST" returns here, and then jumps to UN.
*					
S2	0	'EA2'	BCU/1	UC	If successful, "TEST" returns here. Link to "EA2."
*					
	3	'F'	LKRG		(Post instruction) Save return address.
*					
F	2	EAHI	CARH		
	3	EALO	CARL	R	Access the branch address in core.
	3	EALO	UCOM		Route "EALO" to bus 2.
	2	UCOM	ALL	INC	Increment "EALO"
	3	ALO	EALO		and save in EALO.
	0	'G'	BCU	CCL	If no carry EA now ok for second half-word of branch address.
*					
	2	EAHI	ALL	INC	Carry--need to increment EAHI.
	2	ALO	EAHI		Save incremented EAHI.
G	1	CDR	UCOM		Route core data to bus 3,
	3	UCOM	OP2LO		and save operand 2.
	2	EAHI	CARH		
	3	EALO	CARL	R	Access 2nd half-word of branch address.
*					
	0	'JUMP'	BCU/1	UC	Branch (after 1 post instruction) to "JUMP."
*					
	2	CDR	OP2HI		Get 2nd half word of effective address.
*					
UN	2	ICHI	CARH		
	3	ICLO	CARL	R	Access next half-word of program.
	3	ICLO	UCOM		Route ICLO to bus 2.
	2	UCOM	ALL	INC	Increment ICLO,
	3	ALO	ICLO		and save.

Label	Bus	Source	Destination	Function	Comment
	0	'IFCH'	BCU	CCL	If no carry, "IFCH."
	2	ICHI	ALL	INC	Carry set, increment ICHI
	0	ALO	ICHI		and save.
	0	'IFCH'	BCU	UC	Now go to "IFCH."
*					
*					
*					
*					"STH" uses "EA2" to calculate the
*					effective address and then completes
*					the STH instruction, returning
*					control to "IFCH."
STH	0	'EA2'	BCU/2	UC	Link to "EA2" to compute effective
*					address of op 2.
	3	'H'	LKRG		(Post instruction) Save return
*					address.
H	1	IRO	SHR	R4	Right justify R1 field.
	2	EAHI	CARH		Load high-order byte of effective
*					address into core address buffer.
	3	'000F'	ALR		Load mask.
	0	SHR	ALL	AND	Zero all but R1 field.
	3	ALO	SHR	L1/0	Double R1 to get scratch address.
	1	SHR	SAR	R	Access (R1).
	2	SODR	UCOM		Route (R1) to bus 1.
	1	UCOM	CDR		Load (R1) into core data buffer.
	3	EALO	CARL	W	Load address, and do the memory write.
I	3	ICLO	UCOM		Route low order half-word of
*					instruction counter to bus 2.
	2	UCOM	ALL	INC	Increment low-order half-word of
*					instruction counter.
	0	'J'	BCU	CSET	If carry, jump to "J" to increment
*					ICHI.
	3	ALO	ICLO		Otherwise, save the incremented
*					instruction counter
	0	'IFCH'	BCU/2	UC	And return to "IFCH" (after 2 post
*					instructions).
	2	ICHI	CARH		(Post instruction) Load high-order
*					byte of instruction counter into
*					core address buffer.
	3	UCOM	CARL	R	(Post instruction) Load low-order
*					half-word of instruction counter
*					and access.
J	3	ALO	ICLO		Save incremented ICLO.
	2	ICHI	ALL	INC	Increment ICHI,
	2	ALO	CARH		And load it into core address buffer.
	0	'IFCH'	CU/2	UC	Return to "IFCH" after (2 post
*					instructions).
	2	ALO	ICHI		(Post instruction) Save incremented
*					ICHI.

Label	Bus	Source	Destination	Function	Comment
	3	UCOM	CARL	W	(Post instruction) Access next
*					half-word of program.
*					
*					
*					
*					"ST" completes execution of the ST
*					instruction, then accesses the next
*					half-word of the program and
*					returns to "IFCH."
ST	1	IRO	SHR	R4	Right justify R1 field.
	3	'000F'	ALR		Load mask.
	0	SHR	ALL	AND	Zero all but R1 field.
	2	EAHI	CARH		Load high-order byte of effective
*					address into core address buffer.
	1	ALO	SHR	L1/0	Double R1 field to get scratch
*					address of R1.
	2	SODR	UCOM		Route (R1) to bus 1,
	1	UCOM	CDR		and load into core data buffer.
	3	EALO	CARL	W	Load low-order half-word of effective
*					address into core address buffer and
*					write.
	3	EALO	UCOM		Route EALO to bus 2,
	2	UCOM	ALL	INC	and increment.
	3	ALO	EALO		Save EALO.
	0	'K'	BCU	CCL	If no carry, go to K.
	2	EAHI	ALL	INC	Carry, so increment EAHI.
	2	ALO	CARH		And load into core address buffer.
K	0	'I'	BCU/3	UC	Jump (after 3 PI's) to I.
	2	SODR	UCOM		(P.I.) Route SODR to bus 1.
	1	UCOM	CDR		(P.I.) Load (R1) into core data
*					buffer.
	3	EALO	CARL	W	(P.I.) Load low-order half-word of
*					effective address to core, and write.
*					
*					
*					
*					"EA2" computes the effective address
*					for operand 2 in RX format
*					instructions. It assumes IRO,1
*					contain the instruction and LKRG
*					contains the return address.
EA2	2	'0000'	EAHI		Initialize EAHI to zero.
	1	IR1	ALR		Load IR1 into ALU,
	0	'OFFF'	ALL	AND	and zero all but D2 field.
	3	ALO	EALO		Store this in EALO.
	2	'F000'	ALL	AND	Now zero all but B2 field,
	1	ALO	SHR	L4/EA	and right justify.
	0	'BZ'	BCU	ZSET	If B2 = 0, jump to BZERO.
	3	SHR	SHR	L1/0	Otherwise double B2 to get scratch
*					memory address.

Label	Bus	Source	Destination	Function	Comment
	1	SHR	SAR	R	Access the base register.
	3	EALO	ALR		Load EALO into ALU,
	2	SODR	ALL	ADD2	and add the base register contents.
	3	ALO	EALO		Store this in EALO.
	2	EAHI	UCOM		Route EAHI to bus 3,
	3	UCOM	ALR		and load into ALU.
	2	SODR	ALL	ADC	Get high order byte of base address
*					from scratch, and add it to EAHI.
	2	ALO	EAHI		Store this in EAHI.
BZ	1	IRO	ALR		Load IRO into ALU.
	0	'000F'	ALL	AND	Zero all but X2 field.
	0	LKRG	BCU	ZSET	If X2 field is zero, return to
*					caller.
	2	ALO	SHR	L1/0	Otherwise double to get scratch
*					address of X2.
	1	SHR	SAR	R	Access scratch memory.
	3	EALO	ALR		Load EALO into ALU.
	2	SODR	ALL	ADDZ	And add (X2).
	3	ALO	EALO		Store this in EALO.
	2	EAHI	UCOM		Route EAHI to bus 1,
	1	UCOM	ALR		and load it into ALU.
	2	SODR	ALL	ADDC	Add the carry (if any),
	2	ALO	EAHI		and store in EAHI.
	3	EALO	SHR	R1	Shift the 1 s b of EALO into spill,
	0	'SPE'	BCU	SSET	If spill set - have spec. exception.
	0	LKRG	BCU	UC	Otherwise effective address ready.
*					Return to caller.
*					
*					
*					
*					"TEST" assimilates the condition
*					code from the internal format, and
*					compares it to the mask bits. It
*					returns via LKRG if unsuccessful,
*					and to [LKRG] + 1 if successful.
TEST	1	'2000'	ALR		Load mask corresponding to overflow
*					bit.
	0	CCR	ALL	AND	Load CCR and compare to mask.
	0	'CCB'	BCU	ZCL	If zero bit is clear, then match.
*					Branch CC = 3.
	1	CCR	SHR	L1/0	Shift N bit into spill.
	0	'CC1'	BCU	SSET	If spill set, then CC = 1, so
*					branch there.
	2	SHR	SHR	L1/0	Shift Z bit into spill.
	0	'CC0'	BCU	SSET	If spill set, then CC = 0, so
*					branch there.
CC2	0	'MSK'	BCU/1	UC	CC = 2 by default. Jump to MSK
*					after 1 P.I.
	3	'0020'	T1		(P.I.) Put mask for CC2 into T1.

Label	Bus	Source	Destination	Function	Comment
CC3	0	'MSK'	BCU/1	UC	
	3	'0010'	T1		(P.I.)
CC1	0	'MSK'	BCU/1	UC	
	3	'0040'	T1	UC	(P.I.)
CC0	3	'0080'	T1		
MSK	3	T1	ALR		Load the condition code into ALU.
	1	IR0	UCOM		Route IR0 to bus 2.
	2	UCOM	ALL	AND	Compare condition code with M1 field.
	3	LKRG	UCOM		Route LKRG to bus 0 and bus 2.
	0	UCOM	BCU	ZSET	If zero bit is set, then
*					unsuccessful, so return to (LKRG).
	2	UCOM	ALL	INC	Otherwise increment (LKRG) and
	0	ALO	BCU	UC	return there.
*					
*					
*					
*					"LINK" has 2 global entry points;
*					"LINK" and "JUMP." "LINK" assembles
*					PSW <32-63> in R1 and then loads the
*					branch address into the core address
*					buffer and ICHI * ICLO. The access
*					of this memory location is started,
*					and control returned to "IFCH."
*					Conditional branches enter at "JUMP"
*					when a branch is successful.
LINK	1	'2000'	ALR		Load overflow bit mask
	0	CCR	ALL	AND	and check OV bit of condition
*					register.
	3	'CC3'	BCU	ZCL	If OV set, CC = 3.
	0	CCR	SHR	L1	Shift N bit into spill.
	0	'CC1'	BCU	SSET	If N set, CC = 1.
	1	SHR	SHR	L1	Shift Z bit into spill.
	0	'CC0'	BCU	SSET	If set, CC = 0.
	0	'L'	BCU/1	UC	Jump to L. CC = 2 by default.
	1	'2000'	ALR		(P.I.) code for CC = 2.
CC3	0	'L'	BCU/1	UC	Go to L, (1 P.I.)
	3	'3000'	ALR		CC = 3 code (P.I.)
CC1	0	'L'	BCU/1	UC	Go to L (1 P.I.)
	1	'1000'	ALR		CC = 1 code (P.I.)
L	2	T1	ALL	OR	T1 contains instruction length code.
	3	ALO	ALR		
	2	ICHI	ALL	OR	Insert high-order byte of instruction
*					counter.
	1	IR0	SHR	R4	Right justify R2 field.
	3	ALO	T1		Save T1 = PSW <32-47>.
	2	'000F'	ALL	AND	Zero all but R2 field.
	0	ALO	SHR	L1/0	Double to get scratch address.
	1	SHR	SAR	W	Load scratch address register.
	2	T1	SIDR		Write PSW <32-47> into scratch
*					memory.

Label	Bus	Source	Destination	Function	Comment
	2	ICLO	SIDR		Write PSW <48-63> into scratch
*					memory.
*					Linkage now complete.
JUMP	2	OP2LO	UCOM		Route
	3	UCOM	ICLO		
	0	'FF00'	ALL	AND	Zero old instruction counter m.s.
*					byte but save prog. mask.
	1	ALO	ALR		
	2	OP2HI	ALL	OR	Insert new m.s. byte of instruction
*					counter.
	0	ALO	ICHI		Save new ICHI.
	2	ICHI	CARH		
	3	ICLO	CARL	R	Access next word of program.
	3	ICLO	UCOM		Route ICLO to bus 2.
	2	UCOM	ALL	INC	Increment ICLO.
	3	ALO	ICLO		Save incremented ICLO.
	0	'IFCH'	BCU	CCL	If no carry, go to 'IFCH'.
	2	ICHI	ALL	INC	If carry, increment ICHI.
	0	'IFCH'	BCU/1	UC	Jump to 'IFCH', (1 P.I.)
	2	ALO	ICHI		(P.I.) Save incremented instruction
*					counter.
*					
*					
*					
*					"HWSU" does set-up for the half-
*					word instructions. "EA2" is used
*					to compute the effective address of
*					operand 2. "HWSU" transfers
*					directly to the execution routine
*					whose entry point is in LKRG.
HWSU	3	LKRG	T1		Save (LKRG) in T1.
	0	'EA2'	BCU/1	UC	Link to subroutine "EA2" to compute
*					effective address.
	3	'M'	LKRG		(P.I.) Save return address.
M	2	EAHI	CARH		
	3	EALO	CARL	R	Access half-word at EA.
	3	ICLO	UCOM		Route ICLO to bus 0.
	0	UCOM	ALL	INC	Increment low-order half-word of
*					instruction counter.
	1	CDR	UCOM		Route CDR to bus 2.
	2	UCOM	OP2LO		Get operand 2 from core.
	2	ICHI	CARH		
	3	ICLO	CARL	R	Access next word of program.
	3	ALO	ICLO		Store incremented ICLO.
	0	'N'	BCU	CSET	If carry clear, ICHI ok, else go
*					to "N".
P	2	OP2LO	ALL	TRL	Transfer OP2LO to ALO.
	3	T1	UCOM		Route T1 (= execution routine
*					address) to bus 0.
	0	UCOM	BCU/1	NSET	If high-order bit of OP2LO is 1,
*					branch after
	2	'FFFF'	OP2HI		(P.I.) setting OP2HI to all ones
*					(sign extend).

Label	Bus	Source	Destination	Function	Comment
	0	UCOM	BCU/1	UC	(N not set) Branch after
	2	'0000'	OP2HI		Zeroing OP2HI.
N	2	ICHI	ALL	INC	Increment ICHI
	0	'P'	BCU/1	UC	and jump to P. (after 1 P.I.)
	2	ALO	ICHI		(P.I.) Save incremented ICHI.
*					
*					
*					
*					"FWSU" does set-up full for full
*					word instructions. "EA2" is used
*					to calculate the address of operand
*					2. "FWSU" transfers directly to
*					the appropriate execution routine
*					(in LKRG).
FWSU	2	EAHI	CARH		
	3	EALO	CARL	R	Access operand 2.
	3	EALO	UCOM		Route EALO to bus 0.
	0	UCOM	ALL	INC	Increment effective address.
	3	ALO	EALO		And save in EALO.
	0	'Q'	BCU	CCL	If carry clear, EAHI ok.
	2	EAHI	ALL	INC	Otherwise increment EAHI.
	2	ALO	EAHI		And save in EAHI.
Q	1	CDR	UCOM		Route core output data to bus 2.
	2	UCOM	OP2LO		Save low order half-word of operand
*					2 in OP2LO.
	3	ICLO	UCOM		Route ICLO bus 0.
	0	UCOM	ALL	INC	Increment ICLO.
	2	EAHI	CARH		
	3	EALO	CARL	R	Access second half-word of operand 2.
	1	CDR	UCOM		Route core output to bus 2.
	2	UCOM	OP2HI		Save core output in OP2HI.
	2	ICHI	CARH		
	3	ICLO	CARL	R	Access next half-word of program.
	0	LKRG	BCU/1	CCL	If no carry, jump to execution
*					routine.
	3	ALO	ICLO		(Post instruction) save new ICLO.
	3	ICHI	ALL	INC	Carry set - so increment ICHI.
	0	LKRG	BCU/1	UC	Now go to execution routine.
	3	ALO	ICHI		Save updated ICHI.
*					
*					
*					
*					"CMP" does final execution steps
*					for CR and C instructions.
CMP	1	IRO	SHR	R4/0	Right justify R1 field.
	3	'000F'	ALR		Load mask.
	2	SHR	ALL	AND	Zero all but R1 field.
	0	ALO	SHR	L1/0	Double to get operand 1 scratch
*					address.

Label	Bus	Source	Destination	Function	Comment
	1	SHR	SAR	R	Access low-order half-word of (R1).
	3	OP2LO	ALR		Load low order half-word of
*					compared into ALU.
	2	SODR	ALL	SUB2	Subtract comparands.
	2	OP2HI	ALR		Load high-order half-word of
*					comparand 2.
	2	SODR	ALL	SUBC	Get and compare high-order half-
*					word of comparand 1.
	0	'R'	BCU	OVCL	If overflow clear go to R.
	1	STR	SHR	L1/0	Overflow set. Shift sign bit into
*					spill.
	0	'S'	BCU	SSET	If spill set, go to S, result < 0).
	3	SHR	SHR	R1/1	Complement N bit by shifting in a 1.
	0	'IFCH'	BCU/1	UC	Go to IFCH after 1 P.I.)
	0	SHR	CCR		(P.I.) Save new condition in CCR.
S	1	SHR	SHR	R1/0	Complement N bit by shifting in a 0.
	0	'IFCH'	BCU/1	UC	Go to IFCH (after 1 P.I.).
	1	SHR	CCR		(P.I.) Save new condition.
R	0	'IFCH'	BCU/1	UC	Go to IFCH (after 1 P.I.).
	1	STR	CCR		(P.I.) Save new condition.
*					
*					
*					
*					"LOAD" completes execution of
*					LH.L. and LR instructions. Control
*					is returned to "IFCH." Operand 2
*					is located in OP2LO * OP2HI.
LOAD	1	IRO	SHR	R4/0	Right justify R1 field.
	3	'000F'	ALR		Load mask into ALU.
	2	SHR	ALL	AND	Zero all but R1 field.
	0	ALO	SHR	L1/0	Double R1 to get scratch address
*					of (R1).
	1	SHR	SAR	W	Load address into scratch address
*					register.
	2	OP2LO	SIDR		Load OP2LO, and begin write cycle.
	0	'IFCH'	BCU/1	UC	Return to "IFCH" (after 1 P.I.).
	2	OP2HI	SIDR		Load OP2HI.
*					
*					
*					
*					"ADD" executes the addition required
*					for AR, AH, and A instructions. The
*					condition is updated, exceptions
*					checked and control transfered to
*					"LOAD" upon completion of "ADD."
*					The code for handling fixed point
*					overflow immediately follows this
*					code as indicated, but has not
*					been implemented.

Label	Bus	Source	Destination	Function	Comment
ADD	1	IRO	SHR	R4/0	Right justify the R1 field.
	3	'000F'	ALR		Load mask.
	0	SHR	ALL	AND	Zero all but R1 field.
	2	ALO	SHR	L1/0	Double to get scratch memory
*					address of R1.
	1	SHR	SAR	R	Access low-order half-word of R1.
	2	OP2LO	UCOM		Route OP2LO to bus 3.
	3	UCOM	ALR		Load OP2LO into ALU.
	0	SODR	ALL	ADD2	Add (R1) low order half-word to
*					OP2LO.
	0	SODR	UCOM		Route R1 high-order half-word to
*					bus 1.
	2	ALO	OP2LO		Save low-order sum in OP2LO.
	1	UCOM	ALR		Load R1 high-order half-word into
*					ALU.
	2	OP2HI	ALL	ADC	Add with carry OP2HI.
	0	STR	CCR		Store the resulting condition code.
	2	ALO	OP2HI		Save high-order sum in OP2HI.
	0	'CMK'	BCU	OVSET	If OVSET, jump to "CMK" to check
*					program mask.
	0	'LOAD'	BCU	UC	Go to load to complete this
*					instruction.
CMK	1	'2000'	ALR		Load mask.
	2	ICHI	ALL	AND	Load program mask and check OV bit.
	0	'LOAD'	BCU	ZSET	If clear, go to load.
FPOV					Otherwise process fixed point over-
*					flow exception.
*					(Unimplemented)
*					
*					
*					
*					"SUB" executes SP, SH, and S
*					subtractions similar to "ADD."
*					Code is identical to "ADD" except
*					with SUB2 and SUBC arithmetic
*					operations in place of ADD2 and
*					ADC. In the case of an overflow
*					exception, control is transferred
*					to FPOV above. The code is not
*					repeated here for brevity.

BIBLIOGRAPHIC DATA SHEET	1 Report No. UIUCDCS-R-73-564	2.	3. Recipient's Accession No.

4. Title and Subtitle

A Modular Microprogrammed Computer with Concurrent Decentralized Control

5. Report Date: January, 1973

6.

7. Author(s)

Mark Loren Ketelsen

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Computer Science Department
University of Illinois
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

US NSF GJ-36265

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D.C.

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This thesis presents a computer organization capable of concurrent processing at the microinstruction level. A complete logical description is given, and the results of simulation studies are presented. An emulator for the IBM System/360 is developed, and its performance analyzed for comparison with various models of the IBM line.

17. Key Words and Document Analysis. 17a. Descriptors

Computer Organization
Microprogramming
Emulation

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release unlimited	UNCLASSIFIED	119
	20. Security Class (This Page) UNCLASSIFIED	22. Price

MAR 13 1973

OCT 29 1973

BOUND BY
HM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.559-564(1973
Modular microprogrammed computer with co
3 0112 088400566